A Programmer's Environment for Music Analysis

by

Donncha Seán Ó Maidín, M.Sc., B.Mus.

Submitted in May 1995 to

The National University of Ireland

for the degree of

Doctor of Philosophy

and revised in September 1995.

This thesis was registered in

The Department of Music, **Faculty of Arts**,

University College, Cork,

and was completed under the internal supervision of

Professor David Cox

and under the external examination of

Professor Anthony Pople of the University of Lancaster.

Table of Contents.

Table of Tables

Table of Figures

Acknowledgements.

I would like to acknowledge many people who helped, over the space of more than two decades, with the realisation of this work. Firstly I would like to mention two people who provided help in the early stages and have since died. Professor Aloys Fleischmann was my first supervisor. His energy and dedication proved an inspiration. The second person was Breandán Breathnach who was always forthcoming with help by making material available. Thanks are due to Professor Nick Sandon, who supervised my work for over a year, and to Professor David Cox for his valuable assistance at a crucial stage of the work and for supervising its completion. Thanks to my wife Deirdre, for her constant support during the project, and for help in proof-reading the manuscript. There are a number of people and bodies at the University of Limerick whose contribution I would like to acknowledge. Support from Professor Kevin Ryan and Dr. Seamus O'Shea was always forthcoming in facilitating my work. Thanks for advice on layout must go to Dr. Richard Sutcliffe and Professor Tony Cahill and also to Dr. Gareth Cox of Mary Immaculate College who made valuable recommendations on an early draft. Thanks also to Professor Micheál Ó Suilleabháin for help with the work in Chapter 6. Help in the production of the manuscript and for locating reference information was always forthcoming from Norah Power, Hilary Kenna and Patty Puch. Thanks also to Gemma Ryan for doing the final proof-reading and to the University of Limerick for financial support.

I am indebted to Professor Anthony Pople for suggesting many improvements which I incorporated into this revised version.

Chapter 1. Introduction.

1.1 Overview.

This thesis demonstrates the feasibility of a software environment for the general processing of representations of music scores. The proposed score representation is at a level of abstraction that is appropriate for musicological purposes. In particular it is suitable for analysis. Henceforth this representation and its software environment are referred to as **scoreView**.

The focus of this thesis is on providing a building block that is suitable for use in many areas of computational musicology. The field of computational musicology is characterised by a diverse range of studies that has enlarged the scope of musicological endeavour over the last two decades. In an article by Bernard Bel and Bernard Vecchione[1] these areas of endeavour are characterised as focusing on "the anthropological kernel of musical action", on "the task environment of music" and on "human music processes" as a subset of "human cognitive processes at large". This has resulted, according to the authors, in the emerging of greater autonomy and methodological relevance for "compositional, improvisational and performing and mnesic/perceptive activities". Musicologists are faced with the problems of "merging of unifying domains of knowledge, techniques and practices, which are scattered, and to some extent, disparate". Bel and Vecchione claim that

> "The challenge of a new cognitive-oriented musicology will be to establish a relevant close bond between sciences and techniques applied to music; sound and intelligence engineering; formal, experimental, historical and hermeneutic sciences; anthropological and action sciences; and the philosophies of aesthetics, praxis and cognition.
>
> In all these domains of musicology, theoretical computer science is playing a crucial role dealing with problems of knowledge acquisition and representation. Over the last decade, the computation paradigm has been brought to the front of the stage, thereby deeply affecting the practice of music and musicology and allowing the emergence of a new (transdisciplinary) domain: computational musicology"

Over these twenty years of computer-based musicology, projects tended to originate with specific musicological goals. This focusing on task, rather than on tools has had a serious downside that has resulted in the almost total lack of appropriate, usable, music-

[1] Bernard Bel and Bernard Vecchione "Computational Musicology" Computer and the Humanities, volume 27 (1993), pp.1-5

theoretic software tools and of computer standards for music representation. These lacks present a very serious barrier to progress. Projects that deal with one area of computer-based musicological endeavour that is concerned with bodies of music represented in staff notation, have suffered immensely from this lack of basic tools. These areas of endeavour are referred to as corpus-based musicology.

In order to do general processing of representations of music scores, it is necessary to have two representations for each score. A file-based encoding forms the permanent record of the score in the computer. The main focus in file-based versions is on the issue of representation. A second consideration in the file-based version is ease of encoding. In order to facilitate processing however, there must also exist a distinct musicologist-programmer's version, with a focus on access as well as on representation. Very little effort has been made to focus on this aspect of score representation. The challenge here is in designing a musicologist-programmer's representation, or view of a score, which helps simplify whatever tasks are carried out on scores in a computer.

This thesis examines this second aspect of score representation. That is, it is concerned with the design of a musicologist-programmer's view of a score representation. The significance of concentrating on this is that it provides a way forward, at a higher and more appropriate level than that involved in file-based score representations. When the dust eventually settles on the evolution of file-based standards, such as SMDL[2], the issue of an appropriate musicologist-programmer's representation will survive as a separate and vital concern, which will underlie any effective use of computers for corpus-based musicology.

The **scoreView** environment presupposes the existence of repositories, or corpora of encoded music scores in computer files. The creation of such corpora is a quite separate task from that of automatic music analysis. The difference arises from both the nature of the work involved in creating and maintaining corpora and in the associated driving goals. Ideally, corpora should be created in accordance with internationally accepted standards, on principles of completeness and objectivity of representation. Standardisation, completeness and objectivity are prerequisites for ensuring corpora reuse.

---

[2] Donald Sloan "Aspects of Music Representation in HyTime/SMDL." Computer Music Journal, volume 17, no.4 (Winter 1993), pp.51-60.

There does not exist any generally accepted standard for score representation at present. Hence the ideal of truly sharable corpora cannot be realised. This is unfortunate, as the effect of having at least one standard for music representation would, in time, give a huge impetus to corpus-based studies in computational musicology. There are in existence a number of candidate schemas that show promise of developing into future standards for file-based score representation. It is likely that at least one standard will emerge within the next few years.

Some authors refer to a corpus of music scores as a database. Stephen Dowland Page[3] says "Any form of stored musical material - from a short melody to a large collection of incipits of a repertoire of complete works - can be regarded as a database". The term database is reserved in this thesis for collections of data, such as corpora that have associated information retrieval software of considerable sophistication, and will not be used in the context of an internal score representation.

The approach to score representation taken in **scoreView** is to structure the computer representation for analysis so as to decouple it from file-based corpora. This decoupling makes **scoreView** independent of any future standard for corpus representation. Future standards can be made compatible with **scoreView** by the development of a single piece of additional software. This is an input translator that converts file-base representations into the internal form used by **scoreView**.

1.2 Contribution of this Study to the Field of Corpus-Based Musicology.

The musicologist-programmer's version of a score should have the following characteristics:

It should be generable automatically from the file-based version.

It should carry an objective and informationally complete version of the score.

It should be modelled in the computer in a sufficiently abstract way that the musicologist-programmer's task is made as simple as possible.

---

[3] Stephen Dowland Page Computer Tools for Music Information Retrieval Dissertation for University of Oxford(Bodelian) 1988, p.56.

The musicologist-programmer's environment should be capable of tackling tasks of arbitrary complexity.

In order to foster reuse of software, it should be in a standard form.

At present all previous systems fall short on at least two of the above criteria. The final criterion listed above is one that all score representations lack. This is because the development of a standard form for both file-based representations and musicologist-programmer's representations of scores are both in an evolutionary phase. Some existing analytic systems such as those by McLean[4] and Brinkman[5] exist in a general computing environment of arbitrary power, but these representations lack a suitable level of abstraction. On the other hand, the most advanced system, by Stephen Dowland Page[6], has been devised purposely with a simplified language for user interaction. The computation power of Page's model is that of a finite state recogniser. Finite state recognisers belong to a class of computational models that is less general that a programming language such as C++. The programming language model of **scoreView** is one in which any conceivable computation, including Page's model, can be specified.

1.3 Goals.

The software environment for modelling scores is designed with a number of goals in mind. These include:

1.3.1 Informational Completeness.

This means that the representation holds all of the basic information content of music scores. Each grapheme of a score is represented in relation to its position in a sequence. Here, completeness of representation of the basic information content does not either include or exclude the representation of graphical information that could readily be generated automatically. Examples of information that do not form part of the basic information content include details of horizontal spacing, line thickness and slopes of beams. The current implementation supports the goal of informational completeness in relation to monophonic scores and for a large subset of polyphonic scores. Proposals are

---

4 Bruce Andrew McLean The Representation of Musical Scores as Data for Applications in Musical Computing Dissertation for State University of New York at Binghamton 1988.

5 Alexander R. Brinkman Pascal Programming for Music Research (Chicago 1990).

6 Stephen Dowland Page, op.cit.

presented in the final chapter for extending the representation to cover all polyphonic scores.

1.3.2 Informational Objectivity.

No interpretation of ambiguous notational entities can be made in the score representation without violating the goal of informational objectivity. For example a decision on whether a curved line is a slur or a phrase mark should not be made within the score representation.

1.3.3 Multi-Level.

The building blocks of software are conceived at a series of levels. The most fundamental level is the first level which is basic in nature. By basic is meant that its prime function is limited to giving access to the entire information content of the score. This basic level deals with entities in score, such as time signatures, key signatures, clefs, barlines, notes and rests. Higher level theoretical concepts such as those involved in harmony, are not allowed to clutter this basic level of representation. A major chord for example, appears in the basic model as an unclassified collection of individual notes, and not as any higher level entity of organisation. The current implementation consists of two main levels, with the higher level containing classes to represent and manipulate various abstractions such as pitch class sets and pitch tuples.

1.3.4 Extendibility of the Environment.

This goal involves the building of software components that encapsulate theoretical concepts not found at the basic level. The environment within which **scoreView** exists is open. Additional components of arbitrary complexity may be created and added to **scoreView** by a user as the need arises. Additionally it is possible to organise the resulting complexity into new levels in the hierarchy of levels, as well as packaging them for efficient reuse by others.

1.3.5 Extendibility of Score Representation.

It is desirable to allow for extending the score representation to accommodate constructs that were not catered for in the original design. Such constructs include representations used in ethnomusicology and in some 20th century music.

1.3.6 Abstraction or Complexity Hiding.

Implementation details of the score representation should not be the concern of the music analyst, who should be free to deal with musical, rather than representational issues. **scoreView** representation allows for all of the information content of a score to be accessed using only two constructs. One of these, the Score object, models the score itself, as a container of more elementary objects such as notes, rests and barlines. The second one, the **ScoreIterator** object, models an iterator. An iterator provides a mechanism for locating details within the score. It is also the main mechanism for information retrieval on the score. Resolution of contextual information, such as the effects of time signature, key signature, bar positions and accidentals, occurs automatically, in a hidden layer of **scoreView,** and relieves the analyst-musicologist-programmer of the tedious book-keeping like activities of scope resolution, which would otherwise distract attention from the analytic task.

1.4 Structure of Score Representation.

The system is designed for use by a musicologist who has learned how to program. The environment is an object-oriented one, in which a score is conceived as an object of class **Score**. Class **Score** itself contains autonomous objects which are members of various other classes, such as **Note**, **Rest** and **Barline**. Automatic analysis is achieved by the development and running of user written analytic algorithms which operate on aggregates of objects that make up a score. These in turn, form the internal computer representation of members of a corpus. Since the user's environment is a general purpose programming language, the user is free to build algorithms of arbitrary complexity. There is also the option of incorporating additional external software components, such as statistical tools and harmony classes, into the environment.

The basic **scoreView** representation is built on a number of helpful classes that provide useful building blocks for score representation but which are not found in C++. These include classes for representing rational numbers whose main use is in dealing with time in a score; sets which are used to store various attribute values that attach to notes; frequency stores, which are useful for cumulating results of analyses; tuples which are useful for storing ordered sets of numbers; and strings for holding textual information. The basic level of score representation gives the user access to the entire information content of the score, with the ability to navigate about the score and to perform searches. Above this basic level, a number of additional music abstractions have

been implemented to demonstrate the capability for hierarchically building support for the activities of an analyst. These include classes and functions for pitch class sets, pitch tuples, a parts expert and various analysis oriented functions. These levels are shown in Fig 1.1.

User activities.	developing of analysis software which typically would include use of lower level components. Users may develop, to arbitrary levels of complexity, classes to support various music theoretical abstractions, for use in analysis.
Additional classes and functions.	classes for pitch class sets (class PitchClasses), pitch tuples(class PitchTuple), parts expert(class PartsExpert) and difference algorithms.
Basic classes.	classes to represent music scores and various entities within a music score. class Score, 'glues' these entities together. class ScoreIterator, is the class through which most of the processing is done.
Classes on which **scoreView** is built.	sets(class Set), rational numbers (class Rat), strings (class String), frequency stores (class FrequencyStore), tuples (class Tuple).

Fig.1.1 Relationship between the various classes that are used in **scoreView**. The shaded part is the **scoreView** kernel.

This thesis also demonstrates some ways in which **scoreView** may be used. The corpus used in this study is one of Irish folk dance music encoded in ALMA.[7] The vast

[7] Murray J. Gould and George W. Longemann "ALMA: Alphameric Language for Music Analysis." Barry S. Brook Musicology and the Computer, Musicology 1966-200: A Practical Program (New York: The City University of New York Press 1970), pp.57-90.

majority of the scores in this collection are totally monophonic, with the occasional representation of double stopping providing the only exceptions. In the design of **scoreView**, the goal of designing a system that is capable of representing harmonic and polyphonic music was ever present. It is demonstrated that **scoreView** can be used to represent and process harmonic and polyphonic music, although analysis of such scores does not form part of this thesis.

1.5 Structure of this Thesis.

1.5.1 The process of undertaking corpus-based musicology on a computer is examined by breaking it down into a series of steps which deal with issues such as corpus creation, with the strategy involved in selection of an encoding scheme and with the issues involved in the creation of an internal representation. Factors inhibiting the development of corpus-based musicology are discussed, as well as the prerequisites for progress. A series of encoding schemes that are candidates for a score representation standard are discussed as well as the potential for building a musicologist-programmer's representation of a score using any object-oriented approach. The centrality of such a representation is demonstrated.

1.5.2 Surveys of a series of music systems that use score representations for various purposes, including non-analytic applications such as music printing, sound synthesis and computer aided composition are presented. Five systems for music analysis are surveyed in greater detail.

1.5.3 The goals of this project are then discussed in detail. The various formalisms on which the computer model of the score is built are discussed. These include algorithms, functional abstraction, abstract data types, data analysis and object-oriented approaches to design and programming.

1.5.4 The score is examined from the musician's view of its information content, and the ground is laid for modelling a computer representation. First the physical score is examined and proposals are made for a corresponding musicologist-programmer's view of the score. The score is viewed from a number of aspects. The score itself is conceived as a container object whose components are in the form of separate objects such as notes, rests and barlines. These contained objects are viewed as being structured in time on the basis of horizontal and vertical contiguity. Proposals are made for dealing with scoping

relations within the score, such as the effects of clefs and time and key signatures. Following focusing on the linear structure of notes and rests in a score, a construct called a score iterator is proposed and its central functions in relation to information retrieval and to navigating within the score are discussed.

1.5.5 Detailed descriptions of the classes, functions and types of **scoreView** are given in appendix 1.

1.5.6 A set of applications which demonstrate the use of **scoreView** to build programs for music analysis is given. The first group of these focuses on the use of the system to check the claims of a musicologist. These applications can be programmed, tested, and run in a matter of a few hours by a competent musicologist-programmer with a knowledge of **scoreView**.

1.5.7 A second set of analytic examples are given which demonstrate the potential of **scoreView** for carrying out investigations on a corpus. These applications include:

Scale Finding which shows how we can find the types and frequencies of scales that are used in the corpus.

Feature Extraction which involves extracting and organising information about a melodic feature of double jig tunes.

Melodic Difference which illustrates how we may construct algorithms to estimate the melodic difference between two segments of music. A number of developments of the basic algorithm are discussed and some of these implemented. A proposal is made for further work including the fine tuning of these algorithms.

Form and Exhaustive Search are the fourth and fifth examples which illustrate ways in which a melodic difference algorithm may be used to extract meaningful information from the corpus. One example is concerned with an evaluation of 'crude' melodic forms present in the corpus, and is followed by an example

which involves exhaustive searching of the corpus for identifying exact copies and close tune variants.

1.5.8 A range of further projects are proposed both for the development of basic **scoreView** and for the development of user-specific tools, database application, applications in cognitive musicology, and the development of an expert system for harmony and for mode identification.

1.6 Achievements.

1.6.1 This study demonstrates the feasibility of a computationally viable model of a music score within an environment which has arbitrary computational power, bounded only by the size and speed of the computer hardware, which at the same time, provides an appropriate level of abstraction for use by musicologists.

1.6.2 The design demonstrates the appropriateness of an object-oriented environment for score representation.

1.6.3 The design demonstrates the benefit accruing from conceiving of a score representation in terms of two main classes, one of which models a score and the other, an iterator on the score.

1.6.4 The overall design underlying the C++ implementation is general enough to be implemented in a range of programming languages that support object-orientation, and is usable with a range of the file-base encoding schemes.

1.6.5 The design has potential as a prototype candidate for a future musicologist-programmer's standard for score representation.

Chapter 2. Corpus-based Musicology.

The first section contains an outline of the structure of corpus-based projects for music analysis. The final section deals with prerequisites for the development of corpus-based musicology.

2.1 Corpus-based Music Analysis.

Corpus-based music analysis is characterised by

1. The existence of encoded music scores in machine readable form
2. The existence of software to process these
3. The activities of musicologists who use 2 to process 1 for music analytic purposes.

The prerequisites steps for the creation of corpus-based musicology are represented in the flowchart of Fig.2.1. The flowchart reflects the many steps that historically have been part of analytic projects. It also serves to highlight the enormous potential for the deflection of energies of researchers into tasks that should be avoidable. On the other hand it also serves to highlight those areas where developments offer hope for the future of corpus-based musicology.

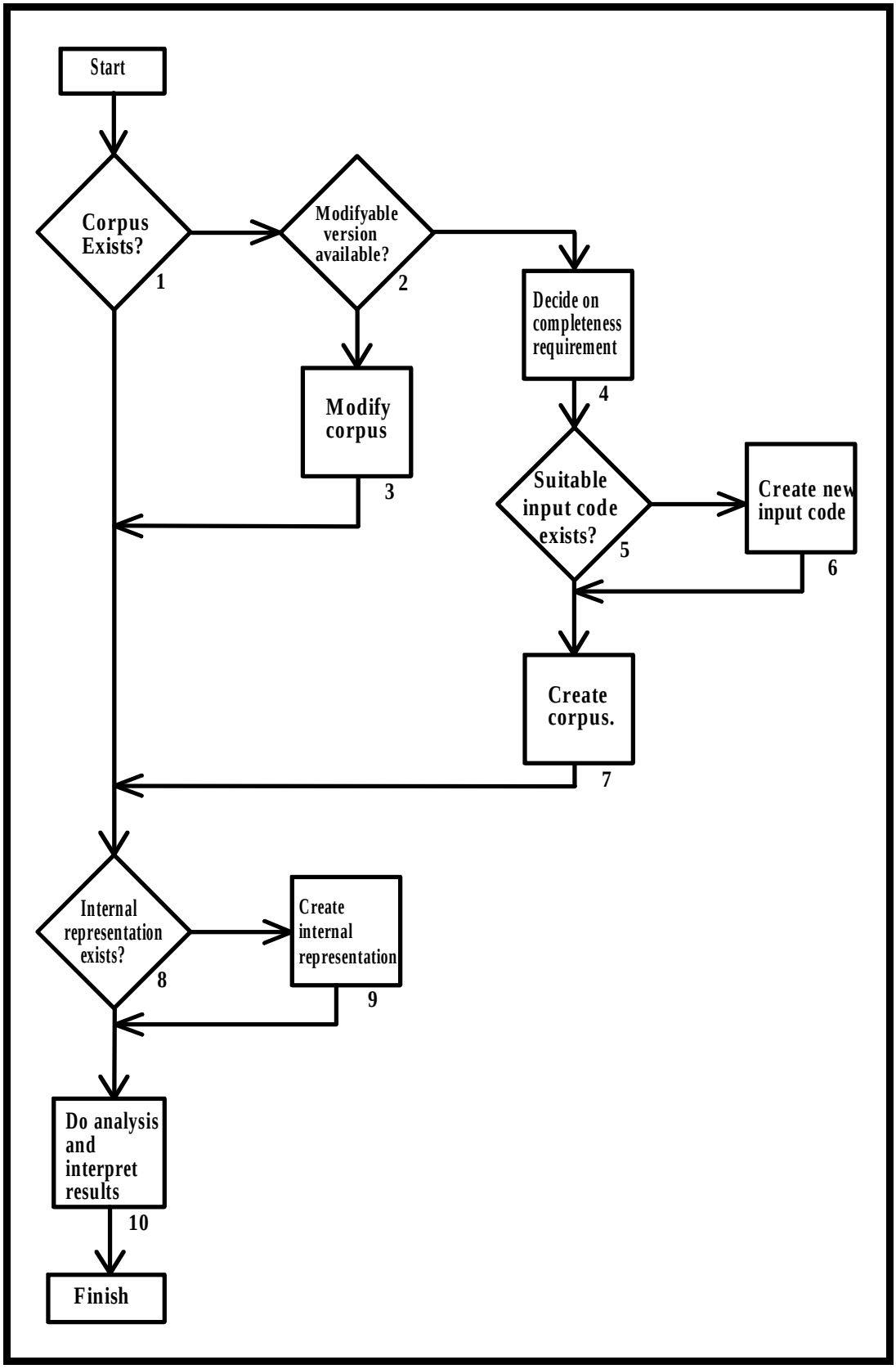


Fig.2.1 Steps in corpus-based musicology.

Step1. If the corpus already exists, the musicologist is in the happy position of being relieved of an enormous amount of effort, and can proceed immediately to step no.8. Furthermore if a suitable internal representation exists, the researcher is able to progress to step no.10, the only one in the entire flowchart that is concerned essentially with the task in hand. This last situation is the ideal one. It will hopefully become commonplace in the future, as it enables the musicologist to concentrate on musical issues, to the exclusion issues of corpus creation and score representation.

Steps 2 and 3. Although a corpus suitable for analysis may not exist for a particular study, it is becoming increasingly commonplace that machine versions of music exist in other forms, and particularly, in forms generated for printing purposes. This raises the prospect of adapting material from the file-based representation used in printing programs for analytic purposes.[8] At a minimum, the utilisation of these sources would involve the acquisition or development of software. This software either transforms the file representation used by the printing package into a form suitable for the analytic software, or, alternately enables the analytic software to access the printing system's files directly. There are, however at least two obstacles that may prevent a totally automatic use of such sources. One of these obstacles arises from constraints imposed by the commercial software developers of music notation systems and the other is intrinsic to the score representation itself. The first obstacle arises from the practice adapted by many manufacturers of notation packages of using a proprietary file representation for scores, while treating this representation as a company secret.[9] In the case of one major package, Finale versions V2 for Macintosh computers, much of this structure has been

---

[8] Stephen Dowland Page, op.cit., p.12.

"Music printing systems are of much greater importance to the musicologist than merely being a convenient way to produce musical examples. As music typesetting techniques reach an acceptable level of sophistication, music publishers can begin to use computer techniques to typeset large repertoires of music. This may mean, as we shall see in a later chapter, that the musicologist could gain access to large amounts of music in a machine-readable form - music which has been entered for typesetting purposes may be used for many studies, producing information of musicological significance."

[9] Many of these products are packaged as 'black boxes' as far as score representation is concerned. Exceptions to this however include Leland Smith's Score, which allows the user to create alphanumeric input for its main features. The alphanumeric representation used in Score lacks some of the capabilities of the internal representation in significant ways. For example, the range of ornaments available is limited. Another music printing system, "The Note Processor" uses DARMS. Most music printing programs have the capability of importing and exporting MIDI versions of their internal representation. However MIDI file format loses too much of the information content of a score to be of general use for corpus-based musicology.

identified by careful detective work. This work was done by F. Nelson in his efforts to use Finale in conjunction with computer aided composition software. The resultant frustration and some inkling of the associated problems can be gleaned from the footnote.[10] The fault may well not totally rest with the reluctance of Coda, Finale's manufacturer, to release details of the format.[11] Attempts to reuse files from printing projects will fail unless the code structures are known in detail. However, even where access to the structure of such codes is available, there can arise further complications which stem from fundamental differences between the natures of score encodings for printing and analysis. It might be argued that the encoding for music printing purposes encapsulates all the basic information content of the music. This is obvious, as a human reader can access the realisation of this information from the resulting printed version. It must follow, surely, one would argue, that the code contains all the information, either implicitly, or explicitly, and consequently is suitable for use in computer analysis. This argument is true in most respects, but falls down in those areas where considerable

---

[10] F. Nelson Music-Research Digest volume 8, no.16 (Thu, 10 Jun 93). Under the heading 'Another Finale Diatribe' , Nelson, whose motivation for this investigation lay with his work in computer aided composition, reports

> Finale's "Enigma Transportable File" (EFT) have the potential to be a kind of music PostScript. An ETF contains every bit of information about a Finale score, including both graphic features and performance details if you choose to specify them. The format of this file is cryptic ("enigma," get it?) but readable by humans if you have enough insight and patience. Writing a file in ETF format is child's play compared to writing a program for algorithmic composition. The catch? Coda doesn't (won't) publish the format.
>
> They fear the loss of commercial advantage and they have a lot of other vague misgivings that still don't make sense after more than four years of talking and writing to them about it. They fear that publishing the format will reveal secrets about their methods of data organisation. In fact, any first year CS student who has had a course in data structures will recognise a multiply-linked list in the "events" structure with lots of messy (but easily mapped) links to the "details."
>
> I have decoded about 85%-90% of the ETF format in somewhat more than three years of sporadic hacking. The intensity of my efforts increases in proportion to the size and complexity of the project at hand. Several large works for wind ensemble have provided the chief motivation for my code-breaking. I can now do quite a lot of what I want to do by directly manipulating elements of an ETF with a program I have written in APL.
>
> Can I publish my methods? Is what I am doing "reverse engineering?" I don't know. If Coda is so block-headed about sharing this important capability with the musical world they purport to serve I suppose they would be equally hard-nosed about my efforts to get my job done even in the face of their hindrances. I would much rather continue to write music than defend against lawsuits.

[11] The fact that Coda has changed ETF format between different releases, and between the Mac and PC versions, might be accounted for by lack of maturity in the representation.

human knowledge is required to interpret the written score. One trivial instance of this is when the human reader reads textual entries, such as title, name of composer and possibly tempo indications at the start of the score. Here the reading brings considerable linguistic and domain knowledge into play in identifying these items of text. Some interpretations are immediately obvious to the human reader, as for example, in identifying which text at the head of a piece of music is the title. In most cases the human score reader will solve the problem of identifying the title even when the text is in an unknown language. Consider now the case of a music analysis program that uses the code from a music printing system such as Score. If the program is requested to print out the name of the piece being analysed, too great a burden would be placed on the software to extract this information in an unambiguous fashion. What is required here with an unmodified printing file, is for the program to determine which of the textual entries is the title. There is a simple solution to this problem. This involves the tagging of this information in the original representation in a form that the printing program ignores, if such is possible, but that is used by the input component of the analysis system. A more substantial problem is tackled by McLean[12] in relation to encoding of polyphonic music in DARMS. He proposes a solution by introducing an additional construct, the **EffectiveDuration** code, into DARMS, to specify the precise duration of notes in cases where the human reader would infer the duration from the context. An example of a situation in which this arises is illustrated by McLean and involves detecting the presence of unmarked triplets from the context by overriding a strictly literal interpretation of the notation. Inclusion of **EffectiveDuration** code in the score encoding avoids unduly complicating the software. In summary, it is possible that a corpus for analysis purposes can be created from files of code that were originally made for music printing purposes. This will normally involve adding a small amount of extra code to the score representation and writing, or otherwise sourcing, software to transform the printing code into an internal form for processing. The potential for the dual use of scores encoded for printing packages, by reusing them for analysis as well as for printing has rarely been realised, although the reverse step, that of generating printed scores from a corpus is commonplace.[13]

---

[12] Bruce Andrew McLean, op.cit., 1988, pp.58-68.

[13] In the MuseData project at the Centre for Computer Assisted Research in the Humanities in Menlo Park, California, an alternative representation in the form of parametric tables has been used for input to the Score program.

Step 4. In creating a corpus, a decision must be taken on how much detail from the score needs to be represented for the study in question. Ideally all details of the score should be encoded. However because of pressures of time and resources, compromises may have to be made. These arise in cases where the direct goal of the immediate analytic task takes precedence over considerations of generality and reusability. It may be adequate to have a simplified representation of musical pitches and durations for a study, or to limit representation used in the study to stressed notes only. The main problem that such short cuts may give rise to is that if subsequently it is decided that additional structures from the score are important, it may become difficult to make progress. Examples of factors that might become important at a later stage in a project that limited the representation to pitches and duration might include the positioning of barlines or the notating of groupettes. If such a requirement becomes apparent at a relatively late stage in the project, then a substantial amount of backtracking is needed in order to re-edit the corpus and to rewrite the software. One of the worst situations that can arise in this case is where the desired extension to the code cannot be accommodated as an add-on to the original version, but instead involves drastically altering the working schema. The safest way to guard against such happenings is to opt for a complete encoding of the scores in question. Complete encoding of corpora is a prerequisite for the more far sighted objective of their reuse.

Steps 5 and 6. A musicologist will invent a private encoding only if relatively simple features of a score are needed. In other cases, one of the existing codes will be used. For a number of reasons, the option of using existing codes, has a number of associated snags. The main problem here is the lack of stability and universality in these codes. DARMS[14], The Plaine and Easie Code[15], ALMA[16] and MUSTRAN[17] are examples of codes that have not achieved universal standardisation. The Plaine and

---

[14] See Bauer-Mengelberg. "The Ford-Columbia Input Language" Barry S. Brook Musicology and the Computer, Musicology 1966-200: A Practical Program, (New York: The City University of New York Press 1970), pp.48-52. Also for a relatively recent dissertation, see Bruce Andrew McLean, op.cit., pp.1-10.

[15] Barry S. Brook "The Plaine and Easie Code.", Barry S. Brook, op.cit., pp.53-56.

[16] Murray J. Gould and George W. Longemann "ALMA: Alphameric Language for Music Analysis", ibid., pp.57-90.

[17] Wenker, Jerome. "A Computer Oriented Music Notation including Ethnomusicological Symbols", ibid., pp.91-129.

Easie Code has achieved its widest use in the Repertoire International des Sources Musicale (RISM) project under the leadership of Barry Brook[18]. ALMA code was developed by Gould and Longman by extending its forerunner 'The Plaine and Easie Code' and, although incomplete, had very good macro facilities that make it easy to type into a computer. MUSTRAN facilitated common practice notation extensions for ethnomusicology and was used in music analysis work of Jerome Wenker and Dorothy Gross. The lack of stability of these codes can be illustrated by considering DARMS code, which is the most common code for corpus creation. One basic problem here lies in the fact that the development of a code such as DARMS is not just a once-off task. In fact it is traditional to refer to the DARMS project, as an on-going effort of overlapping research and dissertations since 1963. It is practically inevitable that such a code, designed initially with however much foresight, will reveal ambiguities and inadequacies with greater practical use. Progress may be made by periodically updating the standard code. It is not enough however, that proposals for change appear in PhD dissertations, or in journals. Unless such proposals are supported by a continuing regulatory agency in the form of a highly visible standardisation body, which is acknowledged as such by the music community and which polices developments and approves changes, the result will be the inevitable development of divergent dialects.[19] This lack of standardisation has two consequences. Firstly, the opportunity for creating music corpora as an end in itself is not feasible, and hence no sharable music databases can come into existence that have a sufficiently standard form to be of truly general use. Secondly, the development of reusable software is frustrated.

Step 7. Corpus creation was done by encoding and typing a score either by means of a card punch machine or, more recently, by entering it directly into a computer. This process is time-consuming, tedious and error-prone. Experience with the current project indicates that the input time per bar of corpus, from the initial encoding and keying to the final quality checking, takes an average of the order of 1 minute per encoded bar. Experience also shows that this tedious work is difficult to sustain over several hours, without the error rate becoming unacceptably high. The creation of all but small corpora

---

[18] Rita Benton, "Repertiore International des Sources Musicales", in Stanley Sadie The New Grove Dictionary of Music and Musicians volume 15 (London: MacMillan 1980), pp.747-9.

[19] In a letter to the Music-Research Digest volume 9, no.34 (Sat, 24 Dec 94), Eleanor Selfridge-Field of the Centre for Computer Assisted Research in the Humanities, announced a forthcoming Handbook of Musical Codes, which is to cover several dialects of DARMS.

takes time that can range from a few person-months to many person-years. This work, which is not the job of music analysts, should ideally be delegated to music coders, whose training must include appropriate levels of keyboard skills, the ability to read staff notation, and knowledge of the encoding scheme. Improvements in the productivity of corpus creation can be achieved from the use of graphical user interfaces and MIDI-based tools that have been available since the mid-1980's. The prospect of automatic optical encoding of scores offers hope for the future.[20]

It is essential that the corpus creating task be done under appropriate editorial and quality management. Again, this activity has no direct connection with music analysis. However, many musicologists that use computers for analysis purposes have to undertake the corpus creation task.

Step 8 and 9. Two representations of music scores are required in a music analysis system. The first representation is a file-based corpus. The second representation is one that is based in the main memory of the computer and is used in writing the analytic software. This internal form is created from the external form by a piece of software, an input translator. For processing a score, the internal form is the enabler, not only of analytic work, but also of all manipulations such as playing, printing, GUI interaction and code translation.

Whatever degree of standardisation exists for representing music at the level of the file, practically none exists in representing music scores in the main memory of the computer. One approach is to copy the file-based score representation into main store in an unaltered form. The main problem here is that this form is most unsuitable for processing. The reason for this is that a one-dimensional string of characters is used to represent a basically two-dimensional structure. Using such code imposes constraints on the software that makes it most tedious to work with, as it is counter-intuitive and error-prone. An analogy could be made of trying to play a game of chess by using a long narrow board formed by cutting the rows of a conventional chess board and reassembling them end to end. Relative simple moves on a conventional chess board, such as those of a queen or knight would become extremely difficult to visualise in this linear, one-

[20] See Walter B. Hewlett and Eleanor Selfridge-Field Computing in Musicology volume 9 (Menlo Park: Centre for Computer Assisted Research in the Humanities 1994), pp.109 - 166, which includes the most recent survey by Eleanor Selfrige-Field of current work. It also contains relevant articles by William McGee and Nicholas P Carter.

dimensional representation of what is basically a two-dimensional structure. Some early analytic work concentrated on processing score representations such as DARMS directly in the computer. This is an approach which imposes a substantial strain on working. This arises from the dichotomy between the basically one-dimensional, character string type representation used for scores in files, and the two-dimensional representation of music notation. Additionally, the type of context-dependent scoping information for time and key signatures, clefs and accidentals, which need a substantial programming effort to resolve, becomes more difficult in a one-dimensional representation. Many of the researchers used the Snobol language, which has powerful string handling features. Although Snobol helped parse a one-dimensional string, the fundamental data remained one-dimensional. What is needed for the internal version of a score is a two-dimensional structure in which scoping information is represented and resolved automatically.

An analysis system can be thought of as consisting of three components. First there is the two-dimensional structure itself. Although this structure plays a central role in all processing, it is not of any direct relevance to the user, at least not as far as the internals of its construction are concerned. It should not matter to the user, for example, whether the internal structure is 'glued' together using pointers or arrays. This is as it should be, as the objective of the user is to do music analysis and not computer science. However, the internal representation plays a central, if somewhat invisible role in making the analysis system work. A second component of an analysis system is a piece of software called an input translator, which is required to create the internal representation from the file representation. The input translator processes a file-based score and builds a two dimensional representation in main memory. Again, this software is not something that the user of the analytic system need be directly concerned with. A third component, and the only one that ideally, should involve the music analysts, consists of a musicologist-programmer's view of the music score. This is the public interface through which the analyst extracts and manipulates information from the score. In later chapters the concept of a musicologist-programmer's view of a score will be developed in detail.

Traditionally, because of the absence of suitable software tools for tackling a job of analysis, researchers had to resort to building up their own main store representations of the music under study. In addition to this, the non-trivial software task of building a program to convert from the file-based computer representation to the main-store based representation had to be undertaken. The task of building a computer representation of the music should be thought of as a task, separate from that of building the software for

analysis. The reason for this is the same as that for decoupling the corpus creation task from the analytic one. Ideally, the analyst should be concerned only with extracting and processing information from the score and not with the internal representations involved.

2.2 Factors Inhibiting the Development of Corpus-based Musicology.

It will be clear at this stage that the task of computer based music analysis contains a minefield of complications for the unwary. Among the causes of these are

- lack of encoding standards,
- lack of reusable corpora,
- magnitude of task of creating corpora,
- lack of software,
- difficulty of specifying goals for computer-based analysis,
- difficulty in arriving at accurate estimates of effort for software development.

Most of the above points can be inferred from the preceding sections. The new points introduced here, include the problem of estimation in software development. This problem has proved notoriously difficult in the software industry the general, especially when dealing with new areas of endeavour. If the software development time for a project is being habitually and grossly underestimated, it will inevitably result in demoralisation of the researcher. The initial design on which estimates are based, often proves to be just the tip of the design iceberg. The ratio involved in this metaphor may not be out of place. As Douglas R. Hofstadter elegantly expressed it (in relation to the development of a champion chess playing programme) -

> "Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's law".[21]

The trap for the musicologist here is that of embarking on a project, to find that, after major effort, the potential end result appears at the end of an ever-lengthening tunnel.

---

[21] Douglas R Hofstadter Godel, Escher and Bach: an Eternal Golden Braid (Middlesex: Penguin Books 1980), p.152.

As Walter B. Hewlett and Eleanor Selfridge-Field report[22] in outlining the early history, reports

> "The passage of time, however, introduced certain practical difficulties which had the effect of slowing down or even crippling some of these early projects. In several cases hardware and software environments were changed by administrative decree. This places an extra burden on funding sources which, despite generous support for the start-up phases of various projects, were often less willing to provide ongoing support. In consequence, little actual processing ever occurred in some instances. In others, there were no results of significance. This denouement of the promises of the Sixties had led by the early Eighties to widespread scepticism about computing in music scholarship."

Further surveys of the lack of development are given by Page[23]

> "In the 1960s and early 1970s much - maybe too much - was written of the potential of the computer as the musicologist's assistant and the music theorist's testing-ground.[24] But by now, two decades later, very little of this potential has been realised; rather, there is considerable resistance to use of a machine in humanistic disciplines."

2.3 Possibilities for Progress.

The more basic cause of lack of progress, was the lack of vision of two prerequisites for constructing a system for the representation and processing of music.

The availability of complete and accurate reusable corpora.

The availability of a musicologist-programmer's view of a score for analytic purposes.

There is little evidence that any of the early representations of music scores were created as a result of focusing specifically on the question of how to adequately represent a score for analysis. Instead the designs were driven by the application in question, whether it was printing, or analysis.

---

[22] Walter B. Hewlett and Eleanor Selfridge-Field "Computing in Musicology. 1966-91" in Computer and the Humanities volume 25 (1991), pp.381-392.

[23] Stephen Dowland Page, op.cit., p.2 (footnote is Page's), also see pp.12-21.

[24] IBM's early sponsorship of the humanities - research posts, conferences, publications - led to a rapid growth in interest in the mid-1960s. The literature from 1967-1970 abounds with preliminary reports, progress reports, and partial results; but very few of the larger, more ambitious projects were completed.

2.4 Prerequisites for the Development of Corpus-based Musicology.

The first prerequisite is the creation of various corpora in complete, standardised forms under conditions of good quality assurance. The creation of these should be a once-off task, quite separate from that of analysis. A realisable ideal could be the eventual encoding of all scores that might be of conceivable interest to musicologists.

The second prerequisite involves the development of appropriate software tools for use by musicologists.

2.4.1 Creation of ReUsable Corpora.

The evolution of a mature encoding standard that has wide acceptance and has continuing monitoring by a visible and accepted standards authority is the main prerequisite for the development of reusable corpora. Lack of standardisation leads to the development within a coding system of incompatible variants, with all of the resultant inflexibility.

The goals of an encoding standard should include the complete and unambiguous representation of the basic information content of a score in a form where the information can be readily recognised by computer software. The order of magnitude of the recognition problem should be parsable. It should not have to depend on software that simulates higher human knowledge. The task may involve complex scans, and recognition of context sensitive information, but should not require advanced tasks, such as natural language processing.

The current position on the emergence of standards is not totally without hope. The main sources from which a number of standards may emerge include

National/international standards bodies,

The manufacturers of notation software,

Institutes specialising in long term corpus creation.

2.4.1.1 SMDL.

The Standard Music Description Language (SMDL)[25] has been under development by a committee of the American National Standards Institute (ANSI) since 1986, under the chairmanship of Charles F. Goldfarb, and the vice chairmanship of Steven R. Newcomb. The project was later transferred to the International Organisation for Standardisation (ISO), and the development has reached the Committee Draft Stage(ISO/IEC CD 10743). In a recent letter[26] to the Computer-Research Digest, Steven R. Newcomb describes the standard as having "not too much left to be done". Unfortunately it would appear that the main thrust of this effort has languished under what Newcomb describes as "a continuing lack of understanding and interest on the part of the music and entertainment industries".

2.4.1.2 NIF.

In a letter[27], Gregory J. Sandell gives details of an inter-industry initiative to develop a standard file format for music notation, called Notation Interchange Format (NIF). This is sponsored by Passport Designs and Coda Music Technology. It is claimed to be a non-proprietary format, which will be available with no licensing fees whatsoever to anyone who wants it. It is claimed that NIF's exceptionally thorough design is the product of a lengthy consensus-building process between a diverse group of notation software designers and researchers in the area of music recognition, musicology and computer science as well as expert users and publishers. Associated with the project is a list of eminent named researchers[28] in computational musicology and in notation software

---

[25] Donald Sloan "Aspects of Music Representation in HyTime/SMDL." Computer Music Journal volume 17, no.4 (Winter 1993), pp.51-60.

[26] Steven R. Newcomb "ISO CD 10743 Standard Music Description Language (SMDL)" Music - Research Digest volume 9, no.35 (Wed 18 Jan 95).

[27] Gregory J. Sandell "Music industry gives us a notation format" Music-Research Digest volume 9, no.36 (Fri, 27 Jan 95).

[28] The original working group includes Nicholas Carter of the University of Surrey, Cindy Grande of Grande Software, Wladek Homenda of Musitek, Steve Keller of Passport Designs, Lowell Levinger of Passport Designs, Chris Newell of Musitek, Mike Ost of Passport Designs, Leland Smith of San Andreas Press, and Randall Stokes of Coda Music Technology.

The advisory board includes Dave Abrahams of Mark of the Unicorn, Garry Barber of Temporal Acuity Products, Alan Belkin of the University of Montreal, Raymond Bily of Midisoft Corporation, Mike Brockman of Temporal Acuity Products, Don Byrd of Advanced Music Notation Systems, Inc, and of Temporal Acuity Products, John Cerullo of Hal Leonard Corporation, Daniel Dorff of Theodore Presser, John Forbes of Boosey and Hawkes, Tom Hall of A-R Editions, Greg Hendershott of Twelve Tone Systems, and William Holab of G. Schirmer.

development. The development is described as being in a state of "now coming to fruition".

As this project has the backing of the notation industry, it does seem to hold a good prospect of delivering a standard, although no commitments are given on the time scale involved. If this emerging standard becomes widely established, its adoption by a standards body such as ISO becomes a possibility.

2.4.1.3 MuseData.

The only mechanism of the establishment of real standards out of coding systems is by usage. Such usage could be promoted by having sufficiently large corpora available to users. The MuseData project at the Centre for Computer Assisted Research in the Humanities in Meno Park, California, is a corpora building project that could play a central role in the establishment of such. MuseData is the name of the main score representation. The system allows for translation of scores between a number of alternate representations, including Kern, DARMS, Score and MIDI. Kern is a new standard file representation that is usable for processing with the software system called Humdrum. An impressive number of scores has been encoded including practically all of the major works of J.S. Bach, as well as multiple works of Beethoven, Corelli, Handel, Haydn, Legrenzi, Mozart, Schubert, Telemann and Vivaldi.

2.4.2 Software Tools for Corpus Analysis.

Four components of a software system for analysis are discussed below. The first relates to the general software environment in which the analysis software works, or the 'score view' used by the analyst. The second deals with the multiple representations of the score information that are desirable in a music analysis system. The third component consists of the supporting software available for analysis. The fourth component is that which provides for the reuse of the software.

2.4.2.1 General Software Environment and Score View.

The approach taken in this study is to consider the most fundamental, but nevertheless general tool that gives the musicologist-programmer the highest level of abstraction or complexity-hiding. The environment used by the analyst should be as

simple as possible, while at the same time providing all the power associated with imperative programming and access to all the information in the score.

As **scoreView** exists in a general purpose programming environment, algorithms of arbitrary complexity can be constructed. Apart from software written by a researcher, libraries of supporting classes can be incorporated into the programming environment as appropriate

Abstraction, that is complexity hiding, is one of the driving forces in arriving at a design of a programming environment for a musicologist. As far as possible, the underlying complexity of representation should not be a concern of the user. Here, complexity results from the nature of the notation itself and from the underlying computer representation. As far as is possible, the complexity arising from both of these sources should be hidden from the user. It should not matter to the user, for example, whether the score representation uses pointers with dynamically created linked lists, arrays or some other construct. The user view should depend on close analogies with a musicologist's view of the paper score, rather than with the computer representation of the score. One useful metaphor is found in some of the underlying structures of object-oriented programming and design. Objected-orientation(OO) conceives of autonomous objects that are encapsulations of data and procedures. The internal details of objects are hidden from the outside world, in this case the musicologist-programmer. Communication with these objects is by means of the mechanism of message passing, or invoking member function, to use the C++ terminology. The principles involved here are illustrated by an example. Suppose that we have a score of the second movement of Tschaikovsky's fifth symphony is represented by an object, let us call it **TschaikovskySymphony5-2**. In an OO environment, the object is created in the computer in a form that contains all the essential score information that might be conceivably used by an analyst. This information may be accessed by sending the object a message. We could send messages to it such as

What is the starting time signature?

What is the starting key signature?

etc.

The object TschaikovskySymphony5-2 will have enough built-in functionality to be able to answer questions such as these.

---- getTimeSig ---> TschaikovskySymphony5.2 ----> (12,8)
(message to object) (object) (response message)

In a C++ program this will appear as the following line of code

TschaikovskySymphony5-2.getTimeSig()

Using a programming environment that supports encapsulation and message passing has a number of benefits over the use of non-objected-oriented environments. Encapsulation ensures that it is difficult for the user to corrupt accidentally the internal representation of the score. An arbitrary level of complexity may be hidden by the combination of message passing and encapsulation. For example, the above message, **getTimeSig()** could be implemented as a search for the first time signature on the first stave of the score, an action that would involve traversing through the initial information of the score, past the clef and the key signature and continuing until the time signature is reached. This complexity, and the complexity of how the score is represented in the first place, is hidden from the user. All the user needs to know is what valid messages or member functions to use and of course, to know the format of replies and to be able to interpret what they mean. Here the replies depend for their form on the allowable constructs within the programming language and on the way in which they are interpreted. It will not be clear at this stage how we might use the message-passing metaphor to access the basic music information internal to the score, such as is found in notes and rests. This will form the main preoccupation of chapter 5.

2.4.2.2 Multiple Representations.

Internal representation of score is basically two-dimensional; file representations are one dimensional. File representations may be based on alphanumeric code. Alphanumeric representation codes include those already discussed such as ALMA, and

DARMS. An advantage of such alphanumeric codes is that they can be created directly with a text editor and they can be inspected, although these aspects are becoming less important with the development of GUI and MIDI interfaces. Non-alphabetic file representations are possible using 'flattened' versions of internal representations. File versions that are close to the internal version could optimise input/output time efficiency. A range of possibilities is illustrated in Fig 2.2.

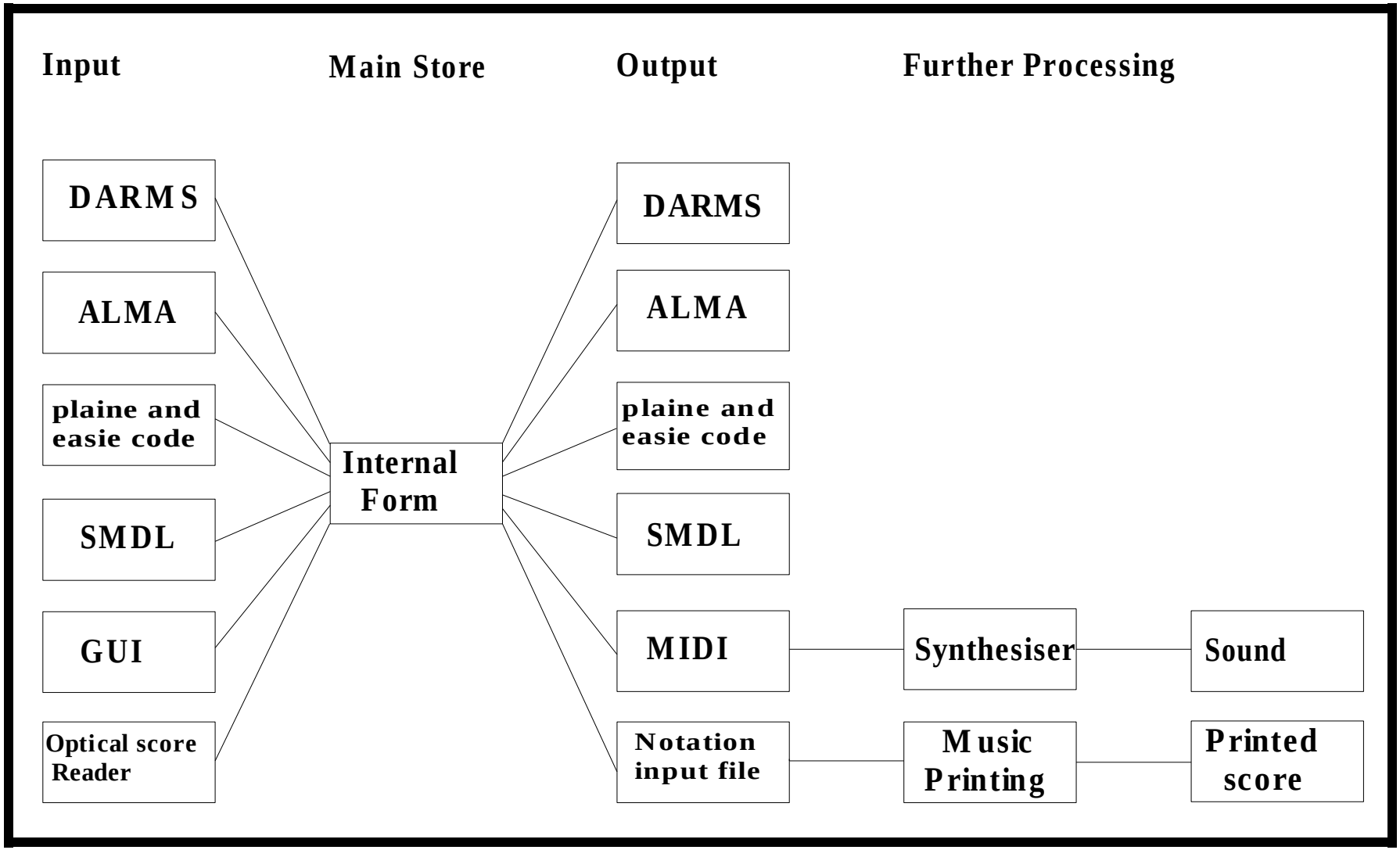


Fig.2.2 Relationships between various representations.

From Fig 2.2, the centrality of the internal representation can be seen. An important feature to note is that the addition of a new external score input representation involves the construction of only one piece of software, an input translator to parse the external code and build an internal representation. Similarly the ability to create a new external representation involves the creation of just one extra piece of software, an output translator, which generates the external form from the internal one. This overall structure enables the creation of an integrated environment for

input of score from a range of different codes,
output of scores in a range of different codes,
hence the ability to translate from one encoding standard to another
by inputting in one code and outputting in another,
processing scores for purposes such as analysis, printing, multimedia, and performing,
the creation of internal form using a GUI/MIDI combination,
the editing of the internal from a GUI/MIDI combination,
the creation of specialised output for input to other packages such as MIDI sequencers and SCORE[29] notation package.

2.4.2.3 Supporting Components.

Any music analysis system will inevitably require additional tools, such as those for playing and printing scores. Playing tools are essential for checking the accuracy of the corpus, and may also be desirable for the building of software for simulating performance by the construction of performer-expert-systems.[30] Printing tools are essential for

[29] The current implementation of **scoreView** has the following components

ALMA to internal form
internal form to MIDI
internal form to SCORE input code

[30] **scoreView** has been used in a pilot project to generate a MIDI stream from a score representation, using mainly rules from Sundberg's research. The project was a final-year undergraduate one for the Computer Science Department at the University of Limerick: Thomas Morrow An Expert System for Performing Irish Dance Music BSc Dissertation for University of Limerick 1993. The research work on which this was based was drawn from J. Sundberg Studies of Music Performance. (Stockholm: Royal Swedish Academy of Music 1983).

checking the accuracy of the corpus and for producing near-publishable printed output. The creation of printing and playing capabilities can be achieved by the construction of two software components discussed in the previous section. An output translator to generate code for a music printing package, and another output translator to generate code for sound synthesis is all that is needed for these capabilities to be realised. The simplest playing may be achieved by translating from the internal representation to MIDI representation that can then be played on commercial synthesisers. **scoreView** contains facilities for generating MIDI output directly, and for the translation from the internal form to MIDIFILE format. Printing is achieved in **scoreView** by the generation of a text file, using an output translator. This may then be used with Leland Smith's printing program Score.

In addition to printing and playing software, a range of supporting software components should be available to facilitate common tasks. The environment fosters the co-usage of other software tools such as databases, parsers, statistical packages and specialised packages in the AI domain.

2.4.2.4 Modifiability/ReUsability of Score Representations.

One important factor in creating a system for score representation is in its ability to develop the software by adding new facilities without disrupting existing capabilities. Because **scoreView** is object-oriented, the powerful mechanism of inheritance may be used for modification and reusability of the score representation. The availability of inheritance supports the creation of new internal score representations that inherit the capabilities of **scoreView**. Examples might include the incorporation of facilities for handling pitch gamuts other than diatonic/chromatic ones.

Chapter 3. Survey of Score Representations and Computer Analyses.

An overview of score representations in systems whose primary purpose is other than music analysis is given in the first section of this chapter. These systems are important in the current study in that they have the potential to contribute either directly or indirectly to analysis applications in a number of ways. This is followed by a section that examines six systems for music analysis.

The next section includes a brief review of systems for music printing whose score representations are generally regarded as having much in common with music representations for analysis. Also included are two less directly relevant areas, both of which have historical and practical connections with score representation for analysis. The first of these is sound-synthesis. One relevance of sound synthesis to **scoreView** lies in its potential to have the computer play the music under analysis. The second area that is covered in this chapter is that of computer aided composition. Again, score representation systems in computer assisted composition share similar techniques to the computer analysis which is concerned with generative studies.

3.1 Score Representation in non-Analysis Applications.

3.1.1 Score Representation in Music Printing.

In the 1950s, the goal of automatic music printing was identified as one that seemed ripe for academic and commercial exploitation. Progress to real usable systems was slow, for a number of reasons. Firstly the task of constructing a printing system from scratch proved to be a substantial one. To be of real use, the output produced has to include the complete complex system of common practice notation. The success of a computer-based music printing system would depend inevitably on its capability for producing printed output of a sufficiently high quality to be of commercial use. Mere novelty and experimentation would be unlikely to impress. Success in this task depended on having stable hardware and software architecture, especially for graphics. Neither of these were

available until the 1980s. Concepts such as that of a graphical user interface(GUI) still had to achieve currency.[31] Additionally printer technology was still in its infancy.

A number of early printing projects emerged, including Donald Byrd's FORTRAN-based printing program SMUT[32], but the project that achieved most attention was started in the early 1960's by Stephen Bauer-Mengelberg and Dr Melvin Ferentz.[33] It was initially called the Ford-Columbia Input Language, but was renamed as the Digital Alternative Representation of Musical Scores. It became widely known under the resulting acronym as DARMS.[34] The early versions of DARMS code appeared in the early to mid 1960s. Typically, instead of employing a music typesetter, a music manuscript is transcribed into DARMS. A DARMS encoded version of a common practice notation score consists of a serial file of alphanumeric characters. The encoding captures the basic information content of a score in enough detail to produce a printed version. Optionally some of the details of the score lay-out could be encoded as part of DARMS. A computer program was written to convert the encoded score from DARMS into instructions to control a photo composition typesetter, which produced the finished printed score and a set of parts.[35] The alphanumeric representation of DARMS survived the original project, and came to be used in

---

[31] The graphical user interface that became commonly available with the development of the Apple Macintosh computer in the late seventies, had its origins in previous systems that originated over half a decade earlier at the Xerox Palo Alto Research Centre.

[32] In the version of SMUT dated 31-May-85 and distributed by Kimball P. Stickney, it is documented as begun in 1968, Smut version 1.1 in July 1975, polyphonic version 2.0 in September 1977, version 2.8 to support shared staves on March 1982. See also Donald Byrd Music Notation by Computer PhD Dissertation for Indiana University 1984.

[33] Bauer-Mengelberg "The Ford-Columbia Input Language" in Barry S. Brook, op.cit., pp.48-52. Also, for a relatively recent dissertation see Bruce Andrew McLean, op.cit., pp.1-5.

[34] According to Bruce Andrew McLean, op.cit., 1988, p.7. The name DARMS was proposed by Melvin Ferents, to honour Edward F. D'Arms, an official who sponsored the project at the Ford Foundation.

[35] According to Walter B. Hewlett and Eleanor Selfridge-Field "Computing in Musicology, 1966-91" in Computers and the Humanities volume 25 (1991), p.386. "..the earliest documented effort at DARMS-related printing was made by Roskin, who implemented both photon and plotter programs as early as 1967 on a code of his own device."

either its original or in a modified form for subsequent printing and music analysis projects.[36]

Early developments in music printing were overtaken by, or developed into commercial products, such as Leland Smith's Score, The Note Processor, which uses DARMS input, Professional Composer, and Finale. The diffusion of the PC and of high quality printing devices resulted in the commercial development of notation packages. Additionally MusicTeX and MuTex[37] and Mutation[38] became available for academic use via Internet.

Most of the later printing systems are designed on the assumption that a purely alphanumeric input encoding would prove inadequate or inappropriate for specifying all visual features to a level required to produce a good appearance. It was found that graphical aspects, such as the positioning of beams, the length of stems, and the shaping of slurs caused problems in the older generation of printing programs. Whereas a computer may be depended on to produce these automatically, a good visual result is not always guaranteed, at least within the capabilities of current software. Nowadays, most score printing systems produce an automatic result, as a first attempt, and then allow the user to modify the resulting appearance by means of graphical editing, using a pointing device such as a mouse. Some of these programs, such as Finale and Professional Composer, dispense completely with alphanumeric input and rely solely on the GUI for score creation, using a combination of mouse and keyboard, and with the optional use of MIDI.

---

[36] DARMS code is used in one of the commercially available printing programs "The Note Processor". According to Hewlett and Selfrigde-Field, ibid., p 387, this was developed by J. Stephen Dydo, a composer educated at Columbia, who undertook to create a DARMS-based music printing program in FORTRAN. It was released for the IBM PC in 1987. See also Walter B. Hewlitt and Elanor Selfridge-Field Directory of Computer Assisted Research in Musicology 1986 (Menlo Park, California 1986), pp.7-34. A number of theses are concerned with music printing and/or score representation in DARMS, of which the most recent one is Bruce Andrew McLean's, op.cit.

[37] See Walter B. Hewlett and Eleanor Selfridge-Field Computing in Musicology volume 8 (Menlo Park: 1992), p.175 for details of the availability of MusicTeX and MuTeX.

[38] Mutation by Glen Diener is distributed by CCRMA at Stanford University.

The form of score representation used in printing systems has a high level of suitability for use in analysis as well.[39]

3.1.2 Score Representation in Sound Synthesis.

As early as the 1950s musicians were investigating the potential of their very limited computer system for creative musical purposes. This happened not only in the academic world, where it might have been expected, but also in the telecommunications industry, where experimentation with music was fostered in a search for understanding human communication. During the late 1950s and into the 1960s impressive advances were made in this area.[40] Sound synthesis systems were developed with a view to using the computer as a creative tool in music composition and performance, by exploiting the newly available theoretical possibilities of digital sound.

The main users of sound synthesis systems were either composers or developers of music instruments. Composers were motivated by creative intent, usually combined with urges to experiment. The development of computer-based sound synthesis meant that for the first time in the history of music, it was possible to synthesise every possible sound, at least in theory. The composers in the analogue electronic music medium in the 1960s had developed an expertise in sound synthesis that was highly constrained by analogue technology. Many of these composers saw in computers the potential to free themselves from the limitations of analogue hardware. In order to generate electronic sound using computers instead of analogue electronics, all that is needed is to specify algorithms for generating the wave forms of the sound, and to program algorithms on a computer for generating and playing these. This contrasted sharply with electronic music practice that was limited by the available analogue hardware, such as filters and sine and square wave tone generators. Another potential that was seized on around the same time arose from the possibility of

---

[39] Bruce Andrew McLean, op.cit., 1988, p.2. also Stephen Dowland Page op.cit., p.iv.

[40] For a historical survey see , Gareth Loy "Composing with Computer - a Survey of Some Compositional Formalisms and Music Programming Languages." Max V. Matthews and John R. Pierce Current Directions in Computer Music Research, (Massachusetts: The MIT Press 1989), pp.291-396. Also for history and techniques of sound synthesis see Charles Dodge and Thomas A. Jerse Computer Music (New York: Schirmer Books 1985). For the techniques of computer-based sound synthesis, see F. Richard Moore Elements of Computer Music. (Englewood Cliffs: Prentice Hall, 1990).

recording sounds with a computer and of modifying them in an arbitrary way. This gave composers in the converging Musique Concrète and Electronic Music traditions, what seemed at the time to be an ideal tool with virtually unlimited potential, although the design of effective algorithms[41] turned out to be a much bigger task than was suspected by the pioneers. Early progress in sound synthesis was rapid, and a real commercial spin-off resulted in the development of mass-produced commercial digital synthesisers in the early 1980's.

The experimentation with sound synthesis systems served to deepen our understanding of some of the processes involved in the creation and perception of music and thus provided a fertile ground for music theory. However the primary goals of this work were not musicological. These differences were reflected in the score representations that were created for sound synthesis. Most computer representations of scores for sound synthesis divide the representation scheme into two parts, the orchestra part and the score part. The orchestra part is involved with expressing sound generating algorithms. The score part consists of a simple structure, a list of notes, arranged one note per line. This one-dimensional format makes it difficult for human readers, who feel more comfortable with music notation through a two-dimensional representation. The task of the score reader is even more awkward to handle when, as is normal in sound synthesis applications, the one-dimensional list is not arranged in time order. The score part was not modelled on the pitch and duration structure of common practice notation. Familiar concepts are expressed numerically, with pitch expressed in Hz, duration in seconds and dynamics in terms of amplitude. Languages of the MUSICx varieties (MUSIC4, MUSIC5, MUSIC11,etc.), CSOUND and CMUSIC are examples of such.

For the designers of these systems, the sheer complexity of common practice notation proved too much of a burden to base an input language for sound synthesis on it. This was partially because composers did not want to be limited by the constraints of common practice notation, which was seen as an inappropriate notation for communicating with a computer and for expressing possibilities in the new medium. The possibility of having common practice

[41] Risset, Jean-Claude Introductory Catalogue of Computer-Synthesized Sounds (Murray Hill, N.J.: Bell Telephone Laboratories, 1969).

notation as a subset of the input code for sound synthesis was addressed by Leland Smith, when he tried to unify the requirements of sound synthesis and music printing. As Loy[42] observes, "... Smith attempted to make a SCORE-like notation for his music printing program, MS, which dates from the same time (the early 1970s), but it became evident quickly that the useful information for synthesis was sufficiently different from that required to typeset a score that the notations had to diverge in nontrivial ways".[43]

Other, more recent composition systems include the object-oriented Common Music[44], the NeXT Music Kit[45] and MODE and SMOKE[46] systems. These systems provide general composition environments with possibilities for real-time interaction, and are not based on common practice notation.

---

[42] Mathews and Pierce op.cit. p.335.

[43] Although sound synthesis systems at present do not use CPN notation, it is possible that future systems may incorporate such a facility. However the orientation of these systems will remain compositional. From the musicologist's point of view, it is important to have access to sound generation facilities. This helps the researcher to detect errors in, and to identify and visualize music from the corpus. Such a facility may be realised by having one component in the analysis software, an output translator, that converts the internal score form into one suitable for input to a sound synthesis system. The simplest form that this takes is by means of a MIDI code output translator that may be used with a commercial synthesiser.

[44] Common music is a high level composition language built on the Common Lisp Object System. It was developed by Heinrich Taube and based on Bill Schottstaedt's language Pla that was developed at CCRMA at Stanford University. See Heinrich Taube Common Music:A Music Composition Language in Common Lisp and Clos in the Computer Music Journal volume 15, no.2 (Summer 1991), pp.21-32.

[45] The NeXT Music Kit documentation is available in machine form from CCRMA at Stanford University.

[46] Stephen T. Pope "MODE and SMOKE" in Hewlett, Walter B. and Selfridge-Field, Eleanor Computing in Musicology volume 8 (Menlo Park 1992), pp.130-2. This is a Smalltalk-based object oriented composition, performance and analysis. See also Stephen Travis Pope "Introduction to MODE: The Musical Object Development Environment" in Stephen Travis Pope The Well-Tempered Object (Cambridge Massachesetts: The MIT Press 1991), pp.83-106.

3.1.3 Score Representation in Computer Aided Composition.[47]

The designation 'Computer Aided Composition' is used for cases where a computer makes some compositional decisions. The main aim is to provide composers with a very much expanded potential for building models for the automatic generation of music. A second potential of these systems lies in the extent to which they illuminate the creative/generative process. Work in computer aided composition began in the 1950s, and was characterised by initial progress which yielded substantial results in the late 1950s and early 1960s.

One of the earliest examples is the Illiac Suite by Lejaren Hiller that was produced as early as 1958[48]. Here Hiller succeeded in using the computer to generate a music score by programming it to select notes randomly, within the rules of species counterpoint. Computer aided composition has been exploited on many subsequent occasions and from various perspectives by composers such as Hiller, Xenakis[49], Laske[50] and Lansky[51]. The score representation that is used in a typical computer aided composition system is radically different from common practice notation. Here musical knowledge is normally embedded as a series of rules. The rules are incorporated into a computer program that generates music. Although the original dynamic for computer aided composition came from experimental music, it is worth pointing out here that its relevance to music theory was identified at an early stage. Generative theories of music structure could be used to build a computer model that generated music. This

---

[47] The first substantial work produced was Lejaren Hiller's Illiac Suite, the score of which appears in the first major publication on experimental music, Lejaren Hiller and Isaacson Experimental Music (New York: McGraw-Hill 1959). See also Loy, op.cit., pp.291-396, and Dodge and Jerse, op.cit., pp.265-322. The IEEE Computer Society has recently formed a Task Force on Computer Generated Music that produces three newsletters every year. For an anthology of developments in computer aided composition, see Denis Biaggi Computer-Generated Music (Los Alamitos: IEEE Computer Society Press 1992). David Cope's Computer and Musical Style (Oxford: OUP 1991), gives details of his own developments in this area.

[48] See Lejaren Hiller Computer Music Retrospective in the series Digital Music with Computer WERGO CD WER 6128-50; see also Lejaren Hiller and Isaacson, op.cit.

[49] Iannis Xenakis Formalized Music (Bloomington: Indiana University Press 1971).

[50] Otto Laske "Composition Theory: An Enrichment of Music Theory" in Interface volume 18 (1989), pp.45-59.

[51] Paul Lansky's Idle Chatter on Wergo CD WER 2010-50.

gave a potential for experimental verification of the generative model by creating music output for validating the model. There is a substantial history of work in this area, most notably those of Baroni, Dalmonte and Jacoboni[52], of Ebcioglu[53] and of Kippen[54].

3.2 Survey of Selected Analytic Systems.

A range of systems is surveyed. These cover a span of over 30 years, and illustrate a variety of approaches for tackling the job of designing an analysis system for music scores.

3.2.1 Michael Kassler's MIR.[55]

The MIR[56] language was developed by Michael Kassler in early 1964 as part of a pilot project concerned with experimenting on ways that a digital computer could assist musicologists in answering internal-evidential questions about a certain corpus of music and in particular, the Masses of Josquin des Prez.

MIR was a specialised computer language for music analysis. It was built on important concepts such as that of a lyne, which corresponds to a part that is performable on an instrument that, at any one time, can produce at most one pitch. There also exists the concept of the current note, that involves the mechanism for making one particular note of the score the current focus of attention, with the implicit notion of a current time as the attack time of the

---

52 Mario Baroni; Rossana Dalmonte; and Carlo Jacoboni "Theory and Analysis of European Melody" Marsden and Pople, op.cit., pp.187-205. also Mario Baroni, Ressella Brunetti, Laura Callegari and Carlo Jacoboni. "A Grammar of melody. Relationships between melody and harmony" in Baroni; and Jacoboni Musical Grammars and Computer Analysis. (Firenze: Olschki 1896). Much of this work was based on earlier work. See Mario Baroni and Carlo Jacoboni Proposal For a Grammar of Melody (Montreal: Les Presses de l'Universite de Montreal 1978).

53 Kemel Ebcioglu, "An Expert System for Harmonizing Chorales in the Style of J.S. Bach" Mira Balban, Kemal Ebcioglu and Otto Laske Understanding Music with AI (Menlo Park: The AAAI Press/The MIT Press 1992), pp.294-334.

54 Jim Kippen and Bernard Bel. "Modelling Music with Grammars: Formal Language Representation in the Bol Processor", Marsden and Pople, op.cit.

55 An account of the nature and history of MIR appears in Stephen Dowland Page, op.cit., pp.73-76.

56 Michael Kassler "MIR - A Simple Programming Language for Musical Information Retrieval" in Harry Lincoln The Computer and Music (Ithaca: Cornell University Press 1970), pp.299-327.

current note. MIR allows only one note to be current at any one time. A set of primitives is provided for moving the current position. MIR allows one to locate the current note at the start of lyne 1 of a selected section, to move forward or backwards by n notes, to move between lynes, to move to a specific measure or to a specific note within a specific measure. Special primitives are included to traverse all the notes of a score. Additionally, primitives are provided for doing comparisons of various entities, for doing arithmetic, for moving data, and for performing output. The main mechanism for retrieving information from the score is via a number of dedicated storage locations consisting of computer words which hold information about the score in general, and about the current note or rest. One such word allows the current note to have a unique identifier that can be treated as a kind of variable.

MIR represented a remarkable achievement for its time, and contained many of the features, if only in embryonic form, which form part of **scoreView**. The language structure for MIR is not high level, and programs resembled assembler code. Each instruction had the general format of (1) a normally optional label followed by (2) the name of the command followed by (3) one or more operands. A sample of MIR is shown in Fig 2.1. It locates the highest and lowest notes, in terms of pitch, in lyne numbered two of the composition being processed.

```
         TOMEAS    1
         TOLYNE    2
ONWARD   COMPAR    REGCL,=14,REST        TO LOCATION REST IF C.N. A REST.
         MOVE      MEASNO,WA10
         MOVE      NOTENO,WA11
         MOVE      REGCL,WA12
         MOVE      NOTECL,WA13
         MOVE      SEMITO,WA14
NEWLO    MOVE      MEASNO,WA15
         MOVE      NOTENO,WA16
         MOVE      REGCL,WA17
         MOVE      NOTECL,WA18
         MOVE      SEMITO,WA19
RETURN   COMPAR    BARLIN,=3, STOP       STOP IF AT DOUBLE BARLINE
         TONOTE    +1
         COMPAR    REGCL,=14,RETURN      TO RETURN IF C.C. A REST
         TRGTH     SEMITO, WA19, NEWHI   TO NEWHI IF ON NEW HIGH
         TRLTH     SEMITO, WA19, NEWLO   TO NEWLO IF ON NEW LOW
         TRA       RETURN                GO TO RETURN
NEWHI    MOVE      MEASNO,WA10
         MOVE      NOTENO,WA11
         MOVE      REGCL,WA12
         MOVE      NOTECL,WA13
         MOVE      SEMITO,WA14
         TRA       RETURN
STOP     CALL      EXIT
REST     COMPAR    BARLIN,=3,STOP
         TONOTE    +1
         TRA       ONWARD
```

Fig.3.1[57] MIR program that locates the highest and lowest notes, on lyne numbered 2.

Twenty consecutive computer words labelled WA1 through WA20 are reserved to the MIR programmer to use as work areas.

MIR represents one of the early giant leaps in imagination, which paralleled similar leaps in sound synthesis and computer aided composition that were made around the end of the 1950s and the start of the 1960s. Whereas the form of the language was low-level, and hence led to rather long programs, it contained the

[57] In comparison with the 29 lines of code above, the scoreView achieves the same in 5 lines of code.

```
ScoreIterator si(s, 1), siHigh = si, siLow = si;
int hiPitch = 0, loPitch = 10000;
while ( si.step(NOTE))
{
 if ( si.getPitch12() > hiPitch ) { hiPitch = si.getPitch12(); siLow = si;}
 if ( si.getPitch12() < loPitch)  { loPitch = si.getPitch12(); siHigh = si;}
}
```

main fundamental concepts for a usable system for general music analysis. The complexity of how music was represented and of how the associated procedures operate is hidden from the musicologist-programmer. Two factors conspired against its general use that have nothing to do with its intrinsic merits. The first was because of the lack of a corpus written in a widely accepted code.[58] The second reason originated in the immaturity of computer software, in particular in language standards. Attempts at improvement of university computing facilities, which kept changing, resulted in great instability in software. Researchers often found that computers were changed without due regard for maintaining portability. When the original computer on which MIR ran was replaced, the MIR system ceased to work.[59]

3.2.2 MUSIKUS at the University of Oslo.[60]

This project started in 1974 with a definition of MUSIKODE, an alphanumeric music input code by Petter Henriksen and Tor Sverre Lande in co-operation with Prof. O-J Dahl. Subsequently Tor Sverre Lande developed the music analysis system as his thesis in computer science. MUSIKODE structures

---

[58] The input language IML (Intermediate Musical Language) was devised by Jones and Howe of Princeton University for the Josquin project. see also Tobias D Robinson, "IML-MIR: A Data-Processing System for the Analysis of Music" in Harald Heckman Elektronische Datenverarbeitung in der Musikwissenschaft (Regensburg: Bosse, 1967), pp.103-135.

According to Hewlett and Selfridge-Field in "Computing in Musicology, 1966-91" from Computer and the Humanities 25 (1991), pp.381-392.

"The need to keep data very compact encouraged the false economy of providing a pitch name without an unambiguous indication of register, which was to be signalled only when changed. In hindsight it was realised that undetected registral errors in the encoded data jeopardised application at every turn. This lapse in the data prevented an otherwise commendable series of design projects from reaching fruition."

[59] Stephen Dowland Page, op.cit., p.28.

"The IML/MIR system developed at Princeton, for example, which was originally designed as a general-purpose music analysis and information retrieval system, was written in a strongly machine-dependent programming language, and when the university bought a new computer - a different model - the task of rewriting all the software was too vast to be undertaken for some time."

Also on page 76, Page reports Kassler as suggesting that difficulties of funding contributed to MIR's lapse into disuse.

[60] Music encoding and analysis in the MUSIKUS system, University of Oslo, Dept. of Informatics/Dept. of Music 1988.

the score representation into a series of hierarchies, the note level being the lowest level. These are combined into chords or parts and finally into sections, such as a movement of a symphony or an act of an opera. This level is called the composition level. At the highest level the compositions are combined into musical entities. This is called the catalogue-level. It would appear from the manual, that coding of the total information content of a score is possible within the general structure of the code, but that various attributes such as dynamics and tempo have not been implemented in the software. The input form of MUSIKODE is converted into an internal form of MUSIKODE by a program called MUS.

The analysis system consists of analytical programs that are written by professional programmers from the Department of Informatics at the University of Oslo. These programs produce an interactive environment that enables musicologists to develop an analysis using a flexible range of built-in facilities. The musicologist runs the program that loads music from the set of pieces in the database. This gives possibilities of performing combinations of actions, including

1. defining horizontal windows for doing thematic or intervallic analysis,
2. defining vertical windows for doing harmonic analysis,
3. defining recursively embedded windows,
4. locating the window,
5. moving the window through the material (scanning) and collecting observations, where moving can be by a fixed interval of time or by a number of changes,
6. defining points of interest (IPs) within a window, on various bases, for example on all notes within the window or on stressed notes within the window, with the concept of guiding parts introduced to facilitate the traversal of two melodic lines in which the changes occur at different times,
7. selecting some from a series of pre-programmed analytical tools that create tables of values, for example TIPS, RIPS, BIPS AND SIFT which produce various pitch class set type calculations on collections of notes (IPs) within a window,
8. saving and restoring tables.

The internal score model as viewed by the analyst has a close analogy with the visual score. The windowing/point of interest metaphor can be readily visualised, and learning to use the system for analysis of a corpus involves considerably less effort than would be the case if the analyst were to learn a programming language. As in all such solutions, this quick road to using a computer for music analysis has the downside of inflexibility when compared with a full programming environment. Only analytic tools developed by the professional programmers are available to the music analyst. The music analyst using MUSIKUS is limited to selecting the actions that take place, such as setting states that influence the effect of subsequent actions, and saving, retrieving and displaying data.

3.2.3 The Essen Computer-Aided Research Project.[61]

This project uses large databases for the purposes of archiving, classification, cataloguing and analyses of ethnomusicological materials. The project was conceived by looking at the possibilities of extending a commercial database package (STAIRS/CMS) to store melodies. A largely text database, ETNO contains over 450, 000 items relating to ethnomusicological sound material, and three further databases, LIED, LIAO and BALL, contain melodic notations. LIED contains more than 4000 German folk songs, LIAO about 1000 Chinese folk songs and BALL(ad) more than 1000 German ballads. The tunes are encoded in ESAC (ESsen Associative Code). The code represents a melody's pitch as a series of scale steps, which are represented by numbers, with the characters '#' and 'b' used for non-diatonic pitches. A cipher notation is used for Asian music with '+' and '-' symbols representing dots above or below the cipher. In ESAC code, scale step 1 is always the tonic. Durations are encoded in terms of the smallest duration in the note, and the underscore is used to denote multiples of this duration, with a single underscore representing twice the smallest duration. An encoding of Arne's, *Rule Britannia!* is given below. The smallest note duration is a quaver.

---

[61] Helmut Schaffrath "The Retrieval of Monophonic Melodies and their Variants: Concepts and Strategies for Computer-Assisted Analysis" in Marsden and Pople, op.cit., pp.95-109.

```
****ARNE
Rule Britannia
Grossbritannien
g0001  08  g  4/4
 3__.3_   4_4__3_   4_.32_1_   -7___
 5__4__   31435_4_  3__2__   1___ //
```

Fig.3.2 ESAC encoded version of Arne's 'Rule Britania!'.

Note that the scheme used here encodes pitches and durations only. Each entry may be analysed by calculating

1. percentage of intervals, in both ascending and descending form,
2. percentage of scale degrees,
3. rhythmic incipit,
4. scale and mode,
5. range, including lower and upper limits,
6. succession of finals of phrases or cadence tones,
7. succession of stressed tones or accent tones,
8. formal analysis of phrases:
    a: comparing succession of pitches,
    b: comparing the succession of durations,
    c: upbeats, phrase wise.

Various calculated values are then stored in a database, and these values may be used as a basis for retrieval. Musical phrases are pre-defined, generally in correspondence with the words. The analysis of phrases uses a concept of 'distance-variant', by counting the number of stressed notes. For example, a phrase is classified as a 'distant-variant' within a deviation of 30% of all stressed notes.

Searches can be made for one or more instances of the stored values in the database. For example, when a new tune is encoded, a search may be made to see if it, or a related tune exists in the database. One could search for all tunes

with the same rhythmic pattern. If this search retrieves a small number of tunes, they can be examined to see if there are similarities. If, on the other hand too many tunes are retrieved, then one could combine the first criterion with another such as a cadential sequence, and in this way retrieve a manageable number of melodies that may be similar. Apart from searching for instances of a specific melody, the system provides a rich environment for exploring stylistic differences between musics of different origins, such as Chinese music and German folksongs.

In this system there is no programmer's model of a music score, apart from the ESAC version of the music, which is limited mainly to encoding pitches and durations of monophonic music. The musicologist's tool here consists of the ability to make musically meaningful retrieval on the database, possibly based on sophisticated strategies according to various criteria.

3.2.4 McLean's System for Score Representation.

The material discussed here is taken from a PhD dissertation[62] of Bruce McLean. The author is concerned with a number of issues, including, notational completeness in encoding music scores in DARMS. With some new language additions to DARMS to resolve ambiguities in vertical alignment, the author devised a method of creating a version of DARMS in main memory for processing. As an intermediate stage in the process of generating an internal version, McLean generates a canonical version of the source DARMS, which is a complete representation of the score, in DARMS code, but converted into a form in which subsequent processing becomes simpler. A number of additional facilities are also developed which include provision for attaching extra, application-specific data to the internal form of the score, such as might be desirable in analysis. McLean gives the following example:

> "For example, all instances of a specified melodic pattern, e.g., a fugue subject, could be located in the Internal Form by a query language such as the one developed by Stephen Page. The query would be defined as a sequence of pitches and/or durations. The data structure returned by the search operation for the query would be a data structure which could be moulded into an attribute plane and attached at the end of the Internal Form. The locations themselves of the melodic pattern would then be stored in the Internal Form. Subsequent

[62] Bruce Andrew McLean, op.cit., 1988.

searches for the same pattern could be carried out by a relatively high-speed retrieval of the attribute plane rather than by an exhaustive search through the whole Internal Form."[63]

In addition to the internal form of the score, a closely related form is generable, the transport version, for transmission across computer networks.

The authors vision of a retrieval interface is significant.

"A basic set of assumptions is being made about the way in which a musical researcher will employ a computer system. First, it is assumed that a researcher who wishes to engage in computer-assisted methods of analysis of musical scores, or who wishes to perform transformations on the representation of a musical score for other purposes such as printing or translation to an alternate representation, will write an application program in one of the common programming languages (e.g., Pascal, Modula-2, or C). Second, a general-purpose retrieval interface will be made available as a means of searching for and extracting data objects from the internal form. The retrieval interface is a library of functions, or subroutines, which may be called from the researcher's application program. The purpose of the retrieval interface is to serve as a software translation layer between high-level requests made by an application and the complex body of information in the internal form. The retrieval interface presents to an application-writer a readily understandable and limited set of requests, or commands, for the selection of data objects within the internal form. The value of the retrieval interface is that in presenting a sufficiently useful virtual view of the score object and its organisation, it also conceals details of the internal form that would distract the application-writer from his primary purpose. Third, the application itself or additional layers of software services employed by the application, but not the retrieval interface will be responsible for the recognition of patterns in the internal form."[64]

Significantly, McLean is here promoting the principle of abstraction, that is of complexity-hiding.

"Organization to support direct retrieval of objects (notes, chords, slices, measures, and, within analytic applications, voices, instances of themes or rhythmic motives, and others) is required. Types of movement which must be supported are (1) direct access of objects; (2) sequential step-wise traversal, in which the traversal step beyond any particular object could be taken in any of several different directions; and (3) automatic search; there must be a well-defined linear path through the entire Internal Form which may be followed by a search engine. An organisational requirement imposed by thematic indexing applications is the ability to store and access a large number (up to tens of thousands) of small excerpts - the incipits - of scores."[65]

---

[63] ibid, p.176.

[64] ibid p.159.

[65] ibid p.163.

Overall the McLean thesis has at its heart the representation of a musical score for processing. The analytic side however, is not developed in this thesis. One analytical approach by Page that was developed using McLean's representation is given in the second next section. Overall McLean's approach has the integrity resulting from concentrating on the issue of completeness and objectivity of score representation. Time will tell whether the number of extra facilities provided is worthwhile. These facilities include the transport version of the score and the ability to attach application specific data. The ability to extend the representation by attaching extra information coupled with the above quoted suggestion of attaching analytic information such as themes for use in information retrieval suggest that the score representation is verging on a database. This approach runs the risk of becoming top heavy, by using the score representation to support a database, rather than by using database technology for storing such relations.

3.2.5 Brinkman.

One of the most comprehensive attempts to make computers available to musicologist-programmers is in Brinkman's 963 page book "Pascal Programming for Music Research".[66] It has been used by Brinkman for graduate students in music and in seminars for Ph.D. candidates in music theory. The book can be looked at from a number of aspects. First it resembles an introductory computer science book in programming which deals with representing data and procedures in a computer. It deals specifically with the Pascal language. Additionally it contains material that would be covered in an introductory course on data structures and algorithms. It has a liberal set of exercises at the end of each chapter. It differs from a typical undergraduate computer science book in two ways. Firstly, many of the examples and of the exercises are based on music applications. Secondly, there are sections that deal with specifics of musical interest. Among these are a DARMS interpreter, functions for handling pitch class set analysis, and a linked list representation for music scores. This book opens up a world of possibilities for the music researcher, but at the same time it demands that the researcher makes a major effort to come to grips with the material. On a typical undergraduate computer

[66] Alexander R. Brinkman, op.cit.

science course, the computing material would take between two and three semesters for an average beginner to reach the level of expertise necessary for its fluent use. The material is good training for those who want to become developers of software that processes music. However, there is considerable overkill in the effort required from anyone using this book as training ground for computer based music analysis only.

A part of the final chapter in the book is devoted to the design of an implementation of an internal score representation. The score representation is not packaged to a sufficient level of abstraction to be of great use to a musicologist. No attempt is made to hide the complexity of the implementation from the user/programmer. The main implications of this are twofold. Firstly the programmer has to know a lot of irrelevant details, that is, details that are inessential to solving a music problem, in order to use the material. Secondly, the error proneness arising from the possibilities of accidentally modifying some of the internals, is significant. Errors that may arise from the user of the software accidentally modifying any of the pointer values that abound in the representation. Such programming errors are notoriously difficult to detect. **scoreView**, on the other hand uses an object oriented approach that solves most of the problems associated with Brinkman's approach.

3.2.6 Computer Tools for Music Information Retrieval by Stephen Page.

A PhD thesis[67], by Stephen Dowland Page, developed at the University of Oxford demonstrates the feasibility of a music information retrieval system. The user interface is based on a non deterministic finite state recogniser, and has an associated simple language, of the type used in advanced text editors. When the user specifies a search criterion, the system interprets the user specification and uses it to search the database of music scores for matching instances. Some examples of the search criterion are given in Fig.3.3.

the note sequence D, E, F.	D.E.F	(1)
the above in the octave from middle C	D4.E4.F4	(2)
the above with time values of quaver, quaver, crotchet	D4/8.D4/8.D4/4	(3)
the same melody in any transposition	N/8.+2/8.+2/4	(4)
a bar commencing with a quaver rest followed by a crotchet	%0% R/8.N/16	(5)
any sequence of at least 3 Gs	G.G.G+	(6)
any ascending fifth followed by a descending third, ignoring any intervening rests	N.R*.+7.R*.[-3,-4]	(7)
any number of successive notes that do not belong to the key of D major	[^S,E,F#,G,A,B,C#]	(8)

Fig.3.3 Sample search criteria as regular expressions proposed by Page.
Reproduced by permission of S. D. Page.

The name of a note is specified by its alphabetic letter with optional accidental and octave registers. Examples 1 and 2 illustrate the use of pitch specifications by means of letters either without or with octave registers. The '.' represents concatenation. The letter R is used for a rest and N is used for any note. Note and rest duration are encoded as a '/' followed by a number (8 for eighth notes, 4 for quarter notes, etc.), as in example 3. Rising relative pitches may be specified by a positive integer representing the number of semitones in the interval, as in example 4. Falling pitches are specified by negative numbers.

[67] Stephen Dowland Page, op.cit.

The construct **%0%** in example 5 is called an anchor, and specifies that the search should take place at the start of a bar only. The symbols '*' and '+', when they directly follow a construct, are used to specify any number of consecutive occurrences of the preceding criterion. The difference between them is that whereas the '+' specifies the occurrence of one or more constructs, the '*' specifies the occurrence of zero of more constructs. Hence the pattern N+ is equivalent to N.N*. Squared brackets are used to specify a single value from a range or sequence. For example [1:4] specifies one of 1, 2, 3 or 4, while the sequence [-3,-4] specifies either a -3 or a -4. Optionality is specified by a following question mark. Hence [B,C]? matches a B, a C or no note. '|' may be used for alteration. Hence F|G matches any F or G. Additionally it is proposed that expressions formed from these constructs may be combined by means of Boolean operators such as 'and' and 'or'.

The thesis demonstrated the feasibility of constructing a useful prototype, which has a number of restrictions. These restrictions include (1) the limitation of retrieval to note pitches and note and rest durations, (2) the limitation of retrieval to scanning single lines and (3) the limitation associated with the anchoring mechanism which allowed for focusing only on a particular position at the start of every bar. It should be pointed out that these limitations are not inherent to the design of the system. They most likely arose from the need to keep the original project within the achievable bounds of a university dissertation.

Figures on the performance of the system when run on a database containing all the preludes and fugues of Book 1 of Das wohltemperierte Klavier are given by the author in Table 3.1, where that searches were run on a single fugue and on all voices of all of the preludes and fugues.

Query	Single fugue		Entire Database	
	Time	Count	Time	Count
N|R	2	776	114	45,703
N	1	733	109	41,424
N/[1:8]	1	615	110	41,424
G.G+	2	5	105	315
N.+4.+3	2	0	111	141
N.R*.+4.R*.+3	3	0	184	141
C.R*.B.R*.A#	3	2	142	13
C+.R*.B+.R*.A#	3	2	165	14

Table 3.1 Performance times in seconds for Page's system.
Reproduced with the permission of S. D. Page.

The times for accessing one tune are well within the limits of usefulness in an interactive system. On the other hand the times for searching the entire collection are not, at least for queries that produce a small number of retrievals. However, these figures are for hardware of 1988. Improvements could be expected from using more recent hardware.

From the point of view of the music analyst, this system offers the prospect of interactive information retrieval that is easy to learn. With the addition of a music oriented graphical user interface, the immediacy of this tool could be greatly enhanced. As well as being a potential tool for computer-literate musicologists, it has the highly significant attribute of being usable by those who are not computer-literate. This ease of use is achieved at the expense of the power of modelling all effective procedures. The computational power of a finite state recogniser is significantly less than that of a Turing machine.[68]

3.3 Summary.

The systems presented here fall into two categories. There are those which provide general access to all the features of a score representation. These include MIR, McLean's and Brinkman's. McLean concentrates on issues of

[68] William A. Wold, Mary Shaw and Paul N. Hilflinger, Fundamental Structures of Computer Science (Massachusetts 1981), pp.341-364.

completeness of representation and does not have an abstract score view, although his thinking is close. He has indicated since completion of his dissertation, that a system for analysis will emerge.[69] Brinkman's approach does not attempt to hide the complexity of his representation. MIR on the other hand, provides a genuine attempt at score representation, but in a form that is dated and with a primitive environment for algorithm development. Both Brinkman's and McLean's approaches are embedded within a programming language. The computing power associated with these environments is general. Such could be used to build other more limited information retrieval systems such as the Essen and MUSIKUS ones. The Essen system and MUSIKUS are examples of the second and substantially different types of system that do not provide facilities for the musicologist to develop general analytic algorithms, but on the other hand offer the ability to manipulate the results of processing using a set of pre-written programs. The big advantage of these systems lies in their potential utility for all musicologists, irrespective of their level of computer literacy. The power of Page's system lies somewhere between the two. It is usable by musicologists with relatively low levels of computer literacy, and has a relatively low learning overhead. In it, users can express search criteria based on a limited language that can be mastered in a short time. **scoreView** requires a deeper knowledge of computing on the part of the musicologist than is required of Page's system. Specifically it requires the mastering of the technique of programming. The benefit to the musicologist of having a general programming environment lies, not in the resulting ease of use, but in the generality and in the potential complexity of the analytic algorithms that it is possible to write.

The following chapters develop the framework within which the musicologist-programmer's view of a score is developed.

[69] Bruce McLean, "An Editing System for Analysis of Musical Scores" in Walter B Hewlett and Eleanot Selfridge Field Computing in Musicology volume 8 (Menlo Park 1992), p.133.

Chapter 4. Goals and Formalisms.

This chapter examines the general goals of the project. It then examines various formalisms with a view to structuring the representation of score in a computer.

4.1 Goals.

The overall goal of this project is to provide a musicologist-programmer's environment for representing music scores in a computer in accordance with the sub-goals of informational completeness, objectivity, extendibility and abstraction.

4.1.1 Informational Completeness.

By this sub-goal is meant that the basic information content of the score is captured in such a way that any question answerable from the basic information content of the printed score is also answerable from the computer representation. By the "basic information content of the printed score" is meant those factors pertaining to an abstract view of the symbols, which contain all the information in the physical score, but exclude information on incidentals. Such incidental information includes the font used, thickness of line and number of bars per line. Two type setters working from the basic information content only, should produce musically equivalent versions, which may look different in various respects. The overall design of **scoreView** supports unrestricted polyphonic scores. This implementation has developed various member functions such as **locate** and **step** for handling a restricted set of polyphonic scores. It supports multi-stave polyphonic scores but there is a restriction on allowable cases of multiple simultaneous notes which appear on the same stave. Multiple simultaneous notes can exist on the same stave in cases where they are not rhythmically independent. Freeing this restriction to allow for the representation of general polyphonic scores is not difficult to achieve. Some further comments on this appear in 8.2.1. To implement this safely would involve carrying out a substantial amount of testing on a variety of polyphonic scores to ensure the reliability of the implementation.

scoreView allows for the representation of a wide variety of the signifiers found in common practice notation. Additional signifiers can be added as required, by using the appropriate structures with **scoreView**.

4.1.2 Objectivity.

The sub-goal of objectivity, means that the computer representation does not commit the user to a specific interpretation of symbols of the written score in cases where ambiguity exists. Hence the distinction between slurs and phrase marks should not be made in the representation, but instead it should be left to the analysis to disambiguate these. The sub-goal of objectivity does not exclude supplying some additional information that may not be overtly present in the physical score. This arises in cases where various liberties have been taken with the notation. Here it may be essential to add information in order to make the score readable by software. An example of this is where groupettes are inadequately represented in the original score. What is involved here is not the resolution of ambiguities, but the unambiguous interpretation of scores written by people who take liberties with the notation.

4.1.3 Extendibility.

The sub-goal of extendibility means that the implementation should be left open to being modified for use in new situations. Here we are concerned with both the extendibility of the analytic environment and of the score representation.

Software components that encapsulate high level theoretical concepts are not found at the basic level. Additional components of arbitrary complexity may be created and added to **scoreView** as the needs arise. This gives users of the system a capability for building arbitrary complex analytic software. Additionally, it is possible to organise the resulting complexity into new levels in the hierarchy of levels, as well as packaging them for efficient reuse by others.

Also it is desirable to allow for extending the score representation itself to accommodate constructs that were not catered for in the original design, such as score representations used in ethnomusicology or in some 20th century music.

4.1.4 Abstraction.

The sub-goal of abstraction means that the representation should aim to achieve the greatest amount of complexity hiding. Abstraction ensures that users of the system are not unduly burdened by issues of score representation. The challenge here is to develop a musicologist-programmer's view that parallels the musicologist's view of the physical

score. This complexity hiding should involve the automatic resolution of scoping contexts, such as are involved in clefs and in time and key signatures.

The most important question to answer in determining an appropriate level of abstraction is not how the score is represented, but instead on what actions the musicologist might want to carry out on it.

A second aspect of abstraction lies in the ability to structure the analytic tasks themselves at a series of levels. The most fundamental level is the first level which is basic in nature. By basic is meant that its prime function is limited to giving access to the entire information content of the score. This basic level deals with entities in score, such as time signatures, key signatures, clefs, barlines, notes and rests. Higher level theoretical concepts such as those involving harmony, are not allowed to clutter this basic level of representation. A major chord for example, appears in the basic model as an unclassified collection of individual notes, and not as any higher level entity. The current implementation consists of two main levels, with the higher level containing classes to represent and manipulate various abstractions such as pitch class sets and pitch tuples. Principles of abstraction can be applied to extendibility of the environment where the organising of complex processing above the basic **scoreView** level is involved.

4.2 Usage.

The main intended use of the musicologist-programmer's environment is by music analysts.[70] Providing such an environment begs the question of how an analyst might use it. In the following quotation, Gareth Loy[71] puts his finger on one of the problems at the heart of the fruitful use of computers for music purposes.

> "As an art form, music has high-level expressive requirements that are extremely difficult to formalize. But computers require formal expression for all problems they address.".

---

[70] **scoreView** could also be used in any area where a representation of the information content of a music score is required. It could be profitably used to produce more specific analytical tools, as well as in computer aided instruction and in multimedia.

[71] Gareth Loy, op.cit., pp.291-396.

scoreView provides only the basic building blocks on which such formalisations may be constructed. The goal of building highly sophisticated analytic systems is facilitated because -

the environment is a general programming one, which gives the analyst the potential for processing scores to any conceivable order of complexity,

software engineering techniques such as hierarchical decomposition can be used to break complex problems down into successively simpler ones, so that the overall task becomes of manageable proportions,

the environment can become a repository for such complex environments, which can be reused, or incorporated into ever more complex systems.

Typical usage of the system might start by an analyst proposing a theory about a particular music genre. Initially this theory may be expressed in a semi-formal way in natural language. Next, a model is built to enable the theory to be tested. The computer model, when run on an appropriate corpus, provides the possibility of experimental verification of the theory. The expression of the model is in the form of some kind of algorithm which is expressed as a computer program. In chapters 6 and 7, examples are given of this process.

A number of topics that are of use in the expression of formalisms are presented in the following sections.

4.3 Algorithms.

The algorithm is the basic formalism that is used for specifying the actions to be carried out by a computer. Knuth[72], lists 5 properties that a process must have in order to be an algorithm. These are -

Finiteness: An algorithm must terminate after a finite number of steps.

[72] Donald E. Knuth. The Art of Computer Programming volume 1:Fundamental Algorithms (Reading:Addison Wesley 1973), pp.1-9
.

Definiteness: Each step of an algorithm must be precisely defined; the action to be carried out must be rigorously and unambiguously specified for each case.

Input: An algorithm has zero or more inputs, i.e. quantities that are given to it initially before the algorithm begins.

Output: An algorithm has one or more output, i.e., quantities that have a specified relation to the inputs.

Effectiveness: An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using pencil and paper.

Two mini illustrative examples of algorithms are given at the end of the next chapter. They are expressed as fragments of C++ programs.

4.4 Functions.

Functional abstraction was developed in the early history of computer science as a way of organising algorithms. Functional abstraction was supported by some of the first high level computer languages, such as FORTRAN and Lisp. Functions provide a structure for the potential hierarchical structuring of algorithms as well as a capability for data hiding. In other words, functions provide a way of hiding complexity. In order to use a function, one has to know only its name, and details of its parameters and return values.

4.5 Abstract Data Types.

One of the first publications to promote the concept of an abstract data type appeared in "Notes on Structured Programming" by C.A.R.Hoare in a book that was published in 1972.[73] Although the term 'abstract data type' was not used in this book, a thorough treatment is given for a range of data structures. Abstract data types(ADTs) are

[73] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Structured Programming (London and New York: Academic Press 1972), pp.83-174.

generalised mathematical concepts of data, which treat data structures in terms of their abstract properties, instead of in terms of how they are constructed.

The most important contribution of ADTs to software environments is that they promote complexity hiding. It becomes possible to deal with a data structure as an abstraction, instead of as a mere collection of pieces of information. ADT's are of great benefit to users of the resultant software because they need not be aware of the underlying complexity in the ADT.

ADT's are implemented in computer languages, such as Pascal, Modula or C as a datatype and a series of functions. An ADT of a first-in-first-out(FIFO) store of integers, or, a queue of integers, might be implemented in C, and used as in the following example which inserts the integers 5, 6 and 7 into the queue

```
QueueType q;
init(q);
put(q,5);
put(q,6);
put(q,7);
```

This approach had a number of weaknesses, some of which have much to do with the language in which the queue is implemented. Some of the most difficult barriers to creating dependable software arise from the potential to misuse ADT constructs. For example, it is possible to pass data other than integers to **q** using the **put** member function, which has been designed to deal only with integers. Also, the user of **QueueType** is not restricted to using functions such as **init** and **put**, as it is also possible to manipulate the internal data in the queue abstraction, with attendant danger of corrupting it. Additional difficulties arise with attempts to reuse such queue implementations.

4.6 Data Analysis.

The score consists of a wide collection of graphemes that we can categorise as being of many types. The desirability arises of structuring these types. We can draw on

techniques that emerged in data analysis in the 1970s for structuring our view of data.[74] The approach developed by Peter Pin-Shan Chen was to define an 'enterprise schema', which he describes as 'a pure representation of the real world'. The techniques evolved three steps as follows, and although they are oriented towards structuring a businesses database, they have a more general applicability.

(1) Identify entity sets of interest to the enterprise, where an entity is a 'thing' that can be distinctly identified. An entity set is a group of entities of the same type. It is the responsibility of the enterprise administrator to select the entity types that are most suitable to his company.

(2) Identify the relationship sets of interest to the enterprise. Entities are related to each other, and different types of relationships may exist between different types of entities. A relationship set is a set of relationships of the same type.

(3) Identify relevant properties of entities and relationships, i.e. define value sets and attributes. Entities and relationships have properties that can be expressed in terms of attribute-value pairs.

Example: We can identify entities such as Score, Note, Rest, Time Signature. There is a relationship between Score and the rest of these entities in that Score acts as a container for entities of type Note, Rest and Time Signature, as well as for other types. We could also envisage Note and Rest entities as having a relationship of vertical and horizontal contiguity with each other.

Attributes of a note could include its letter name, its octave register and possibly an immediately preceding accidental. Note that this categorisation involves the design decision to make the accidental an attribute of the note, instead of giving it status as an entity in itself. Examples of values associated with these attributes might be

pitch letter = C
octave register = 5

[74] Peter Pin-Shan Chen "The entity-relationship model - A basis for the enterprise view of data" Conference Proceedings of the American Federation of Information Processing Societies (1977), pp.77-84.

accidental = #

Associated with each attribute is a value set, which is the set of allowable values. The value set for the pitch letter attribute is { A B C D E F G }.

The entity-relation model focuses on data. Entities have internal states. The ADT, on the other hand focuses on activities that are carried out on data, and hides details of the data from the user. In the next section, we will see how these approaches can be unified in terms of objects. This gives the benefit of being able to modify entities as in the entity-relationship model, and at the same, time capturing its behaviour. A number of additional benefits accrue from this approach.

4.7 Object Oriented Programming.

4.7.1 Encapsulation and Message Passing.

The object oriented approach to programming arises from a re-casting the ADT view of functions that operate on data. The shift of focus involves combining both data and functions as a single entity called an object. This packaging of data and functions is call **encapsulation**[75] and involves hiding the data so that it cannot interact directly with anything external to the object except through the functions which form part of the object. In non object-oriented programming languages actions are carried out by calling functions. This involves passing data to a function in the form of parameters. The function then performs its operations using the data parameters and optionally return a piece of data as result. In the object-oriented approach, an action is carried out by calling a member function of the object. The metaphor used here is that a message is sent to the object. The message takes the form of a function name together with its associated parameters. The object responds by executing the code associated with this message,

[75] According to Oscar Nierstrasz "A Survey of Object-Oriented Concepts" in Won Kim and Frederick H. Lochovsky Object-oriented Concepts, Databases , and Applications (New York: ACM Press 1989), pp.3-21, encapsulation is the main common element in various approaches to object oriented programming in various programming languages.

which typically changes the state of the object in some way, and optionally, sends further messages to other objects. The shift of emphasis is, according to Brown[76]

> ". . . more closely tuned with the way in which we think about entities in the real world; we rarely divorce the concept of what the entity is (i.e. its state) from what we can do with it (i.e. operators with manipulate it)."

The word 'class' is used to describe the type of object. We talk of objects being of a particular class. Object classes are used in a programming language to automatically create or **instantiate** objects of that class.

4.7.2 Specialisation.

Many object classes have things in common in themselves. They may have similar data components, and also have common operators or functions. Instead of having to define each subclass from scratch, in object oriented programming languages we have a mechanism called **inheritance** which automatically structures this superclass/subclass relationship. Subclasses can inherit some or all of the behaviour of the superclass. Additionally new functions and/or data can be added to the subclass. These additional functions can be used to add new capabilities to the subclass or to override some of the functionality inherited from the superclass. The subclass/superclass relation can be applied recursively. Apart from vertical chains of inheritance arrived at in this way, it is also possible in some object oriented environments for a class to inherit horizontally from more than one class. This mechanism is called multiple inheritance.

Inheritance may be used for a number of purposes. As a way of structuring objects, it provides a tool for abstraction. Classes may be constructed at a series of level of abstractions. Inheritance may also be used as an aid to software reuse. A reuse of software often requires modifications. This leads to the existence of multiple incompatible versions. Inheritance provides a mechanism to avoid this divergence, by providing an orderly way to modify classes without having to re-write them. An example of a multiple inheritance structure appears in **scoreView** where the relationship between classes **Duration**, **Pitch**, **Note** and **Rest** is structured in accordance with the inheritance pattern shown in Fig 4.1.

---

[76] Alan W. Brown Object-oriented databases: their applications to software engineering. (New London: McGraw-Hill 1991), pp.18-23.

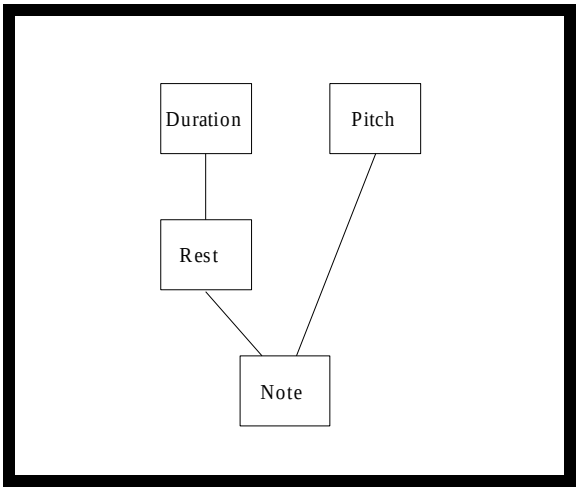


Fig 4.1 Inheritance structure for **Note** and **Rest** objects.

Class **Rest** is built by inheriting class **Duration**. In building class **Note** we reuse both class **Rest** and class **Pitch**. This is an example of multiple inheritance. The effect of this inheritance is that class **Note** inherits all the functionality of both class **Rest** and of class **Pitch**. Hence class **Note** can be queried about what its duration is, because it inherits this member function from class **Rest**. It can also be queried about which octave it is in, as it inherits this member function from class **Pitch**.

In C++ terminology, the superclass is called a base class and the subclass is called a derived class.

4.7.3 Polymorphism and Overloading.

Overloading allows us to attach new meanings to functions and to operators that depend on the context in which they are used. Context can be determined for the functions or operators from the parameter types that they use. Overloading achieves a type of polymorphism, a compile time polymorphism, which enables us to use the same functions or operators in different contexts. Hence in **scoreView**, the function **getPitch12**, which gets the chromatic pitch number, is not restricted to belonging to only one object. It can be called for a **Pitch** object, or for a **Note** object, or for a **ScoreIterator** object. Late binding, which provides for run-time polymorphism when used in conjunction with inheritance, is dealt with in the next section.

4.7.4 Late Binding.

Languages that support inheritance also enables another very powerful facility, that of **late binding** to be handled automatically. This allows us to defer selection of code that acts on an object until run time. This is particularly useful in representing a score that may be thought of as a collection of various objects that are assembled at run time.

An example of the combined advantage of polymorphism and late binding can be seen from the following example. Suppose X is a pointer to an object in a programming language that models any entity in a score. In the non-object oriented version, we will assume the existence of a function called getTag() which returns the type of the object that X points to. The code to invoke various functions to draw X on the screen will look like

```
if ( getTag(X) == NOTE )  drawNote();
else if ( getTag(X) == REST) drawRest();
else if ( getTag(X) == BARLINE) drawBarline();
else if ( getTag(X) == TIMESIG) drawTimeSignature();
else  ..... etc .....
```

The corresponding object oriented construct reduces to

```
X -> draw()
```

which invokes the encapsulated draw function of the object X.

4.7.5 Object Orientation in scoreView.

The choice of an object-oriented representation for a score came about as an evolutionary process. The representation of a score that preceded **scoreView**[77] evolved towards the encapsulation of code within a data object. The conversion to the use of an object oriented language has the significant advantage of providing automatic support for encapsulation. In the previous version, encapsulation was implemented as Pascal function calls. The availability of polymorphism in the new version allows for

[77] Donncha Ó Maidín "Computer System for Music Analysis" Helene Charnasse Informatique et Musique (Paris: ERATTO 1984), and Donncha Ó Maidín "Representation of Music Scores for Analysis" Alan Marsden and Anthony Pople, op.cit., pp.67-93.

considerable simplification in the user interface. The availability of inheritance allows for a natural reuse of software, and in particular the availability of multiple inheritance allows for a better structuring than would have been possible otherwise.

C++ was chosen as the language of implementation. Unlike some other object oriented environments such as Smalltalk, it supports multiple inheritance. C++ is the de-facto industry standard. This means that enough commercial might backs the development of C++ environments to ensure that in most cases they work correctly, and efficiently. Also we can be sure that C++ environments keep pace with developments in operating systems and user interfaces, across a very wide range of machines. C++ runs efficiently on much less expensive hardware than most other objected oriented systems, a fact that is less important now than it was in the past. Implementations of C++ are reasonably portable. Currently the software runs in 3 environments, DOS, Windows3.1 and Unix, using the Borland C++ compiler for DOS and Windows and the GNU compiler on Unix. Many C++ compilers have good interfaces to AI languages and GUIs, which it is hoped to exploit in the future.

Chapter 5. Score Views.

This chapter examines the score from the musician's view of its information content, and lays the ground for modelling a computer representation. The first view involves the physical score, that is the material record of the score. At the other extreme we have the view of the score from the programmer's vantage. The process of unifying these two views centres on forming a sufficiently abstract view of the physical score that encapsulates its basic information content.

5.1 The Score as a semi-formal System of Representation.

In one aspect of music theory, that involved with the representation of music in scores using common practice notation, one might expect to find a formal system. The exigencies of the use of common practice notation for communicating musical ideas among composers and performers might imply the existence of a lingua franca that possess an unambiguous grammar and semantics. Although much of common practice notation approaches this ideal, there are a number of factors that make it unrealisable.

Score notation is derived from common practice. Hence it is not the rules that generate score notation, but instead the other way round. Rules come for observing the common practice in the first place. Such common practice has semi-formal conventions that arise from the needs of communication and invention, rather than a fully formalised underlying structure.

Staff notation has inconsistencies within itself. In a study by Huron[78], he examines staff notation in terms of the signifier and the signified. A clef, the signified, is signified by a symbol, the signifier, located on a stave. One ideal criterion is laid down by Huron is that no two signifieds may share the same signifier. Common practice notation violates this criterion when a sharp sign is used both in a key signature and as an accidental. Although it is possible to distinguish between these two signifieds, in most cases by the context, it is not necessarily so. There exists a similar potential for ambiguity in the notation of slur and phrase marks. Another desirable criterion, that of

[78] David Huron "Design Principles in Computer-based Music Representation" Alan Marsden and Anthony Pople, op.cit., pp.5-39.

reversibility between the signifier and the signified, is not always possible, as for example, in the case of crotchet rests, where two signifiers exist.

There are also many examples where norms of the notation are transgressed. The selection of the time values for notes which form groupettes is one such example.

Score notation itself is not a static thing, it continues to evolve. This yields individual notational solutions that may or may not ultimately become part of the common practice.

In **scoreView**, there is the assumption that the score has been encoded in a manner that captures the basic information content of the physical score. Any ambiguity in the score due to incompleteness of the notation must be resolved at the encoding stage. Notational incompleteness should be made complete as a separate editorial task, prior to encoding. Provision is made within **scoreView** for dealing with specific cases of reversibility, such as that involving crotchet rests.

The following sections deal with the structure of the score from different points of view. Many aspects of the score are discussed here in relation to the physical score as well as in relation to its representation in a computer. The first and second sections deal with the score both as an entity in itself and in terms of the entities contained within it. This is followed by two sections that deal with time and with contiguity relations. Next, there is then a section dealing with scoping relationships. The sense, or the absence of a sense of line is the topic of the following section. The final three sections deal with the score reader, and its computer analogue, the score iterator, and the use of the score iterator in locating and scanning actions within a score.

5.2 The Score Entity.

We can look on the physical score as an entity in itself. Hence we can talk about various kinds of operations on the score, such as playing a score, publishing a score or composing a score. The score itself has a number of attributes. These include its title, name, composer, and if it is a printed score, its publisher.

A musicologist-programmer's version of the score is created in the computer by using a score declaration in the processing program. The score is created from the contents of a file in which an encoded version of the score exists, in one or other input

codes such as ALMA, DARMS or SGML. This is done by in **scoreView** for an ALMA encoding by means of the following declaration.

Score s(filename);

where **s** is the name of the score object and **filename** is a variable of class **String** that contains the name of the score file.

We will now focus on the kind of things we might want to do with the object **s**. We might want, for example, to find the name of the score. This is done by calling the member function **getString(TITLE)**. In a similar way we could ask for other details of the score, such as what key it is in, what the initial time signature is, who is the composer is, etc. Such member functions are called in a similar way.

s.getString(TITLE);
s.getString(CMPSR);
s.getString(KEYSIG);

There is not a lot of things we can do with a score as a whole. Two such actions are of use however, to get the score to play itself on the local MIDI hardware or to draw itself on the screen[79]

s.play();
s.draw();

Most meaningful activities are carried out not on the score as a whole, but instead on the various entities that constitute the score.

5.3 Entities within the Score.

We consider a score, not holistically, but as an ordered collection of its constituent entities. These entities in the physical score are represented by graphemes or groups of graphemes that carry symbolic or iconic, or a combination of symbolic and iconic information. Iconic representation is partially used in the encoding of pitch and of pitch

[79] In the current implementation of **scoreView,** the play but not the draw function is implemented.

movement in time, where the pitch height corresponds approximately to height of a notehead, and the passage of time corresponding to left-to-right note symbol sequencing. Music is arranged on one stave or on a system of staves that appear from left to right, starting near the top of the page. These are repeated to fill the page. Clefs and key signatures appear on the leftmost corner of each stave, but time signatures appear with minimal frequency.

The abstract view of the score corresponds to a view of the physical score that is stripped of features that pertain to its physical realisation, while retaining at least all information of potential relevance to a music analyst. Hence the abstract score is not tied to page layout or to a particular print face and can be viewed as consisting of staves of indefinite length. The physical score has far more clefs and key signatures than its abstraction.

In order to classify the various symbols in the score, it is useful to think of some symbols as having a major status, or as entities, where other symbols may be regarded as belonging to those of major status, or as attributes of the entities.

A list of the main score entities and of their attributes is given below. Most of the entities listed exist in the score as combinations of graphemes. However pitch and durations exist within a score as abstractions, which are useful in the structuring of notes and rests. The term scope is used below either in the context of entities that influence the interpretation of other entities, or in the context of the affected entities. A section on the nature and types of scoping mechanisms is given later in this chapter.

5.3.1 Entity: Key Signature.

Attributes: value, location, open scope.

Value: any one from the 24 possible key signatures, also possibly non-standard extensions.

Default: Key of C.

In the computer, key signature is represented as an object called **KeySig**, with the values of an enumerated type **KeySigType** used to specify the key.

KeySig(keySigType ks = NOKEY);

where

enum
keySigType
{
C, SF, SFSC, SFSCSG, SFSCSGSD, SFSCSGSDSA, SFSCSGSDSASE, SFSCSGSDSASESB, FB, FBFE, FBFEFA, FBFEFAFD, FBFEFAFDFG, FBFEFAFDFGFC, FBFEFAFDFGFCFF, NOKEY
};

The above names are interpretable by treating **S** as standing for sharp and **F** for flat.

F may also denote the note F.

5.3.2 Entity: Time Signature.

Attributes: value, location, open scope.

Value: unnormalized rational number (such as 4/4 or 6/8) , or common time (C, with ancestry in a semi circle) or simple duple time(C with line through it).

In the computer, two classes are used to represent time signatures. The first one,

TimeSigType is used to model the rational number aspect of time signatures.

TimeSigType(long n1 = 4, long d1 = 4);

The second class represents a time signature in a score.

TimeSig(int n1 = 4, int d1 = 4);

TimeSig('C') is used for a common time(4/4) object, and **TimeSig('c')** is used for simple duple time(2/2).

5.3.3 Entity: Clef.

Attributes: value, location, open scope.

Value: French violin, soprano, mezzo soprano, treble, bass, alto, tenor or baritone.

In the computer, a clef is represented as an object called **Clef**, with the values of an enumerated type **ClefType** used to specify the clef.

Clef(clefType c = NOCLEF)

where **clefType** is

```
enum
clefType
{
  FRENCH_VIOLIN, SOPRANO, MEZZO_SOPRANO, TREBLE, BASS,
 ALTO, TENOR, BARITONE, NOCLEF
};
```

The next three entities are represented as character strings in a score. They are created in a score using the overloaded '+' operator of **ScoreIterator** class with class **TaggedText**. They are retrieved with the **getString(const tagType & tt = TAG)** member function of **ScoreIterator**.

5.3.4 Entity: Metronome.

Attributes: value, location, open scope.

Value: duration value = number.

The duration value is expressed in ALMA. An example of a valid entry of 100 dotted quarter notes per minute is

4. = 100

5.3.5 Entity: Tempo.

Attributes: value, location, open scope.

Value: character strings in restricted natural languages, representing unambiguous tempo indications.

5.3.6 Entity: Expression.

Attributes: value, location, open scope.

Value: character strings in restricted natural languages, representing unambiguous expression text.

5.3.7 Entity: Duration.

Attributes: nominal value, number of dots

Nominal value: breve, whole note, half note, quarter note, etc.

dots: ., .., ..., etc.

In the computer representation, durations are represented at two levels. An enumerated data type of C++ is used for the first level, for note values that are represented in common practice notation by a combination of noteheads, stems and positioning. The representation here is simply a mnemonic mapping from normal names.

enum durType { N0, N1, N2, N4, N8, N16, N32, N64, N128 }

Additionally a duration can be modified by the presence of dots. This is modelled as class **Duration**. Class **Duration** is a score abstraction. It is used in constructing classes **Note** and **Rest**.

Objects of class Duration are created by the constructor

Duration(durType d = N4, int dot = 0);

5.3.8 Entity: Pitch.

Attributes: pitch name, octave register, accidental.
pitch name: A, B, C, D, E, F, G.

octave register: an integer, with middle C starting register no 5.

accidental: none, flat, sharp, natural, double flat and double sharp.

In the score as well as in its computer representation, pitch is an abstraction. One of its uses is in the internal structure of class Note. Objects of class **Pitch** may be created using the constructor

Pitch(char pa = 'C', int oc = 5, accidType ac = NOACCID);

pa can have any character in the range 'A' to 'G'
oc is the octave number, with 5 representing the octave upwards from middle C.

accidType is defined as

```
enum
accidType
{
 NOACCID, F, S, N, DF, DS
};
```

where **F** = flat, **S** = sharp, **N** = natural, **DF** = double flat and **DS** = double sharp.

5.3.9 Entity: Rest.

Attributes: duration, marks, location.
duration: see above, as for Duration entities.

marks: various, including fermata and breath mark.

ambiguity: crotchet rests have two signifiers.

Modifications: The effective time for a rest may be modified by groupette scope.

The computer representation of class **Rest** inherits class **Duration**. Objects of class **Rest** are created as follows

Rest(durType d = N8, int dot = 0, Set e = Set())

Parameters **d** and **dot** are similar to the corresponding ones in class Duration. The third parameter, the set **e**, can contain any relevant combinations of **ntAttrType**. These may include **FERMATA**, **BREATH_MARK** and **ALTERNATE**. For a crotchet rest which uses the English notation, like a reversed '7', the attribute **ALTERNATE** is set, and **d** is set to **N4.**

5.3.10 Entity: Note.

Attributes: duration, pitch, marks, location.

duration: see above, as for Duration.

pitch: see above, as for Pitch.

marks: any of the large number of marks that can apply to a note (staccato, various accents and ornaments, dynamics, octave doubling, etc.).

Modifications: the effective time value for a note can be modified by groupette scope. The effective pitch of a note can be modified by key signature scoping or by accidental-within-bar scoping.

Note entities are represented in a score by class **Note**. Class **Note** inherits from class **Rest** and from class **Pitch**.

Note(char pa = 'C', int oc = 5, accidType ac = NOACCID,
durType d = N8, int dot = 0, Set nr = Set())

Parameters **pa**, **oc** and **accidType** are similar to those in the constructor for class **Pitch.**

Parameters **d** and **dot** are similar to those in the constructor for class **Duration**.

Parameter **nr** is a set which has appropriate combinations of

STACCATO, TIE_FROM, TIE_TO, TENUTO, PLUS, FERMATA, BREATH_MARK, COMMA, TREMOLO, TREMOLO_END, GLISSANDO, GLISSANDO_END, SQUARE_NOTEHEAD, DIAMOND_NOTEHEAD, X_NOTEHEAD, OMIT_NOTEHEAD, OCTAVE_UP, OCTAVE_DOWN, OCTAVE_END, ARPA, PIZZ, HARMONIC, COL_LEGNO, PONTICELLO, PED, REL, OCTAVE_DOUBLE_UP, OCTAVE_DOUBLE_DOWN, OCTAVE_DOUBLE_END, TURN0, TURN1, TURN2, TURN3, TURN4, TURN5, TURN6, TURN7, TURN8, TURN9, TURN, SLUR1, SLUR1_UP, SLUR1_DOWN, SLUR1_END, SLUR2, SLUR2_UP, SLUR2_DOWN, SLUR2_END, ACCENT, HEAVY_ACCENT, UP_BOW, DOWN_BOW, LETTER_TR, BAROQUE_TRILL, GRACE_NOTE, BEAM, UP_BEAM, DOWN_BEAM, BEAM_END, REST_ALLIGNMENT, ALTERNATE, PPPP, PPP, PP, PIANO, MF, FORTE, FF, FFF, FFFF, CRESCENDO, CRESCENDO_END, DIMINUENDO, DIMINUENDO_END.

Delimited scoping information is carried as attributes of note and rest objects. The first and subsequent objects bear an attribute which is terminated by a special marker. Such sequences involve one or more of the following pairs.

TREMOLO - TREMOLO_END
GLISSANDO - GLISSANDO_END
OCTAVE_UP - OCTAVE_END
OCTAVE_DOWN - OCTAVE_END
SLUR1 - SLUR1_END
SLUR1_UP - SLUR1_END
SLUR1_DOWN, SLUR1_END
SLUR2 - SLUR2_END
SLUR2_UP - SLUR2_END
SLUR2_DOWN - SLUR2_END,
BEAM - BEAM_END
UP_BEAM - BEAM_END
DOWN_BEAM - BEAM_END
CRESCENDO - CRESCENDO_END
DIMINUENDO - DIMINUENDO_END

The dynamic marks appear only on the notes against which the letters are to be placed. Dynamic scoping is of the open scoping type, and is not handled by the above mechanisms.

5.3.11 Entity: Barline.

Attributes: bar type, mark, location.

bar type: various combinations of heavy and light lines, possibly with a pair of dots arranged vertically at one end or at both sides.

Mark: a fermata, Repeat1, Repeat2, Da Capo, Da Capo .....

Modifications: Depending on the context, barlines may be used to separate bars and/or to separate sections in a score. If a barline occurs before the metrical end of a bar, it automatically represents a section separator instead of the start of a bar.

Barline entities are represented in a computer by class **Barline**, and an enumerated type **barType** is used to specify the kind of bar in question.

```
Barline( barType br = L, int brN = 0);
```

where br is one of

```
enum
barType
{
 CLHLC, CLLC, CLH, HLC, CLL, LLC, CLC, SHORT, INVISIBLE, LL, CL,
 LC, H, L, DOTTED
};
```

Here **C** stands for double dots, **H** for a heavy line, and **L** for a light line. If the score does not distinguish between heavy and light lines in barlines, the **L** should be used. **brN** is the bar number. Any notes before the start of the first full bar of the score are regarded as being in bar 0.

Each barline may have a number of associated attributes, including

FERMATA, DA_CAPO, DA_CAPO_AL_SEGNO, DA_CAPO_AL_FINE, DA_CAPO_AL_SEGNE_E_POI_AL_CODA, DAL_SEGNO, DAL_SEGNO_AL_FINE, REPEAT1, REPEAT2,

5.3.12 Entity: Words.

Attribute: value, location.

value: text in natural language, words of song, libretto, etc.

Words are retrieved with the **getWords(void)** member function of class **ScoreIterator.**

5.3.13 Entity: Texts.

Attributes: value, location.

Value: text in natural language.

Location: linear position, also may be specified as being above or below the stave.

The Class Text entities are represented as character strings in a score. They are created in a score using the overloaded '+' operator of **ScoreIterator** class with class **String**. They are retrieved with the **getString(TEXT)** member function of **ScoreIterator**.

5.4 Time.

Score entities are arranged in a two dimensional structure that represents simultaneity by means of vertical relationships. Time, and the passage of time is represented by horizontal relationships, with left to right corresponding to increasing time in the cases of notes and rests.

In a monophonic score, or in a single monophonic stave, time is accounted for by notes and rests, according to certain principles.

1. Each note/rest has an ending time which is identical with the starting time of the next note/rest.

2. Duration of notes and rests are measured numerically, in rational numbers.

3. An absolute score time scale may be constructed by cumulating the durations of notes and rests from the start of the score. More conveniently this absolute score time scale can be expressed in terms of the number of bars from the start plus a single rational displacement from the start of the current bar. This measure is used in the locating actions in the next section.

4. At the start of a score, an incomplete bar may be found. That is one whose rational duration is less than the time signature. In this case the partial bar is given the bar number 0. Rational displacements of entities within this partial bar are measured as if the bar were a full one. Hence a single eight note anacrusis in 6/8 time is regarded as starting at a location of 5/8 in bar number 0. The length of any full bar in rational units is, of course, the same rational number that is used to denote the time signature.

A point in score time can correspond to multiple entities in a score. Hence the score location specified by 'one half note distance into bar 2' in Fig. 5.1 specifies a time at which a number of entities in the score are current. These include (1) the end of a quarter note rest, (2) the tenor clef and (3) the start of note 'G'. We see from this that the left to right ordering of entities corresponds to simultaneity in time, except when moving across a note or rest, that is characterised by having an infinite number of points in time, delimited by a starting point and a finishing point. Here a point in score time need not necessarily correspond to the start of a note or rest. For example, the score location specified by 'three quarter notes distance into bar 2 corresponds to a point in time during the playing of the note 'G' in Fig 5.1. If the score were a polyphonic one, there would be at least one entity on each stave corresponding to that specific time as well.

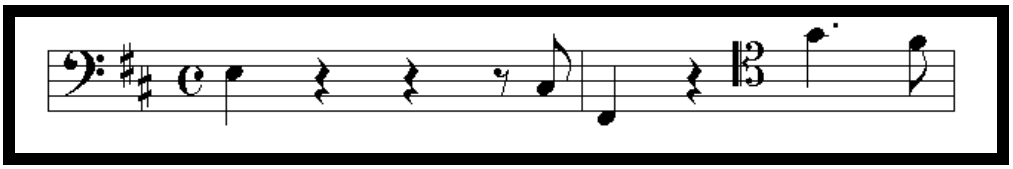


Fig.5.1 Points in score space and score time.

5.5 Vertical Alignment and Contiguity.

In a physical score, vertical alignment corresponds to simultaneity. A definition of vertical contiguity will be made in relation to notes and rests. Notes will be used to illustrate the relationships, but the same principles apply to any mixture of notes or rests. If we consider a pair of notes, and possible score times that relate to them, we can say that notes are vertically contiguous at the score times shown by the red lines in figure 5.2. For example, (b) represents two note that start at the same time, but end at different times. The mirror image of (b), which is not illustrated, corresponds to two note that start at different times but end at the same time.

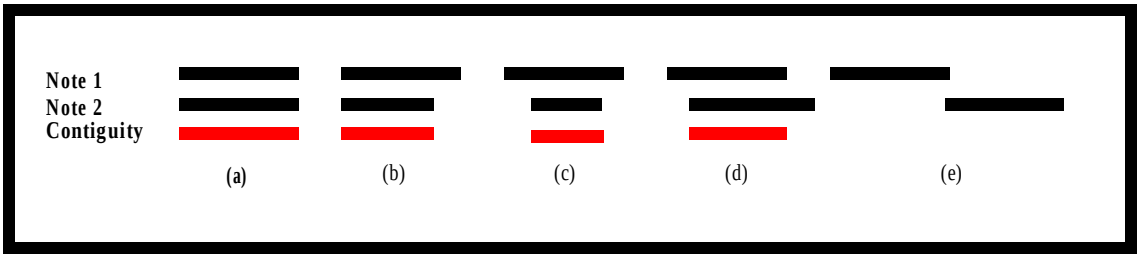


Fig.5.2 Illustration of the possible combinations involved in vertical contiguity.

Note that in case (e) for the coincidence of the end and the start of a note, contiguity is defined in such a way that there is no contiguity in this case.

Simultaneity in score notation uses two basic mechanisms. The first is the mechanism of absolute score time. Notes and rests that are simultaneous share part of the same absolute score time. The second mechanism that is involved is where note onsets are made simultaneous by means of vertical connections, as in Fig.5.3.

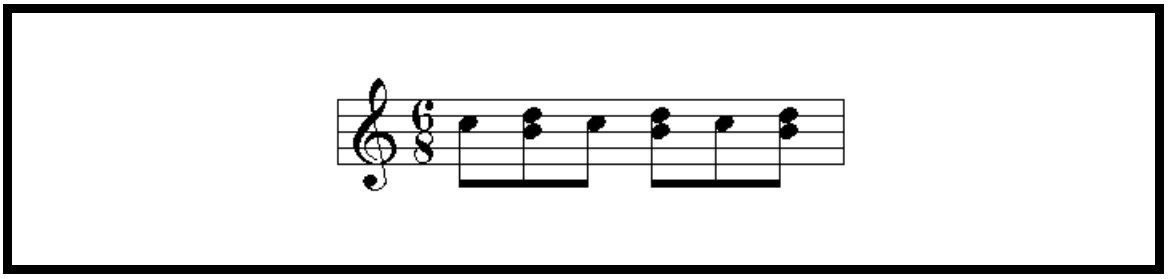


Fig.5.3 Vertical connections.

In certain cases, internal points of interest can be created in notes. This occurs when, during a note, another note has an onset of an offset. For example, cases (b), (c) and (d) in Fig.5.4 contain such points. A musical manifestation of this is where a suspension is resolved. In dealing with standard traversals in section 5.10, we will see that these internal points in the entities are visited.

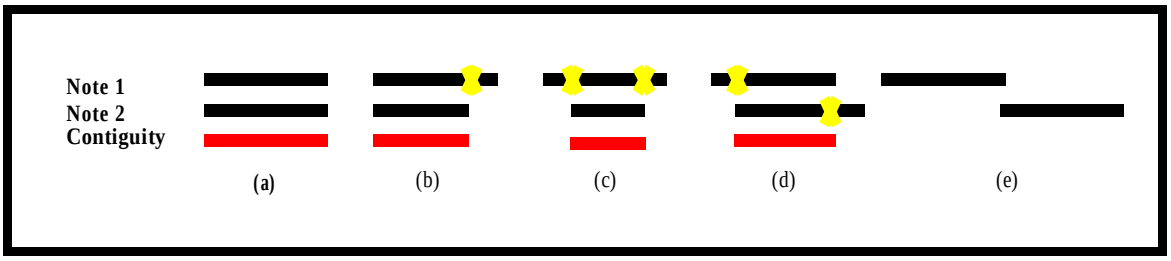


Fig.5.4 Internal points of interest indicated in yellow.

5.6 Scoping Relations.

Certain symbols have an associated scope, that is a range of effectiveness. Here we look at three main scoping mechanisms[80] used in common practice notation. Each stave has its own independent scoping mechanism.

The first type of scoping mechanism, which will be called open scoping, comes into play by means of the appearance of a scope marker. Scope markers have types and values. Two different key signatures, for example, belong to the same type, but have different values. Open scopes remain in effect until cancelled by the appearance of a next scope marker of the same type, or until the score ends. Open scoping is used for clefs, key and time signatures, and metronome, expression, and tempo markings as well as most markings for dynamics.

[80] There is a fourth scoping mechanism. Score attributes, such as title, composer and number could be thought of as having **global scope**. As they are constant for a score, they do not need any special handling.

A second scoping mechanism, called delimited scoping, is explicitly introduced and cancelled. Examples of this are Ped .. Rel, beam start...beam end, octave up .... end octave up, crescendo, with the start marked by the point of the hairpin, and the end by the end of the lines, or if stretched letters are used as in c - r - e - s - c - e - n - d - o , the location of the first c and last o determine the limits of the scope of the crescendo.[81] Delimited scoping is used for calculating the durations of notes within groupettes.

A third scoping mechanism, called bar scoping, is restricted in range to a single bar on a single stave. Scoping mechanisms that operate within the bar include accidental alterations that extend beyond the note on which an accidental is placed. Accidentals can be modified by a preceding additional accidental at the same notated pitch within the bar. All bar scopes expire by the end of the bar in which they are introduced.

Scopes can overlap in a variety of ways. Hence they cannot be represented in the form of a hierarchy. In the computer representation, it is most important that scoping be resolved automatically, in a hidden layer of the implementation. This is done in order to avoid placing too large a burden on the analyst who would otherwise have to calculate scope values. With proper automatic resolution of scoping, we should be able to extract from a note, its pitch and duration information that is calculated by correctly resolving scoping information within its context, due account having been taken of key signature, accidental modifications and groupette membership.

Some scopes of the same type can be nested. Nested groupettes to any level of nesting are theoretically possible, but instances to even one level are rare.[82]

5.7 Sense of Line and Simultaneity.

Scores may differ in the ways in which the identity of lines is present. Choral scores have complete identity of line. A score for a stringed instrument may have a more

---

[81] Crescendo may also have the side effect of introducing a scope of the first type, in that they may effect subsequent dynamics. This is an area where performance practice and artistic interpretation comes into play, and is not modelled in **scoreView**. Hence there is no automatic interaction between dynamic scopes, such as forte or piano, and crescendi or dimuendi.

[82] Instances to one level nesting can be found in transcriptions of a descriptive nature where a high level of accuracy is attempted. One example is in Liam de Noraidh Ceol on Mhumhain (Baile Atha Cliath 1965), p. 53.

complicated sense of line, with occurrences of simultaneous notes representing multiple stopping. In piano scores, the linear and harmonic combinations may reach much higher levels of complexity. The resolving of the complexity of line identity is regarded here as being apart from the task of score representation. What should be represented in a corpus is simply the information content of the printed score, in accordance with the principle of objectivity. It is valid to represent the notes that are present, their values, attributes, whether they have up or down stems and how they are beamed and slurred. The job of tracking two implied lines that are written on the same stave is carried out by another, independent class, separate from the score class. Complexity of linear identity is handled by one aspect of the class **ScoreIterator**. Some simple versions of score iterators are found in **scoreView**. For more complicated cases, the user has the ability to construct score iterators of arbitrary complexity. This constructing may be done either by inheriting the existing class **ScoreIterator**, or by building the new iterator from scratch using the **locate** and **step** member functions of **ScoreIterator**. Some of the issues concerning the design of a score iterator will be dealt with in the section on score traversal, later in this chapter.

5.8 Score Reader.

The simplest case of a human score reader looking into a score, may be modelled as an act of focusing on one entity at any one time. Hence we may conceive of a score reader as having an associated state linked to the entity being viewed.

In studying the score the human analyst will need to be able to locate a particular entity in the score, for example, the first note in bar 100 in the 1st violin line, and to interpret what is read. This will involve, in the first instance, the determining of the key, clef and time signature. Subsequent activities of the analyst may be to scan the notes and rests in a score sequentially along the same stave, or to scan simultaneous notes in some vertical manner. Certainly the analyst will also want to access notes on the basis of some kind of contiguity, and most likely on a left to right basis. This suggests that the analyst's entity-locating activities in reading a score can be encapsulated by means of sequences of operations such as

locate first note in bar 100.
identify clef, and key and time signatures.
step horizontally to next note.

step horizontally to next note.
etc.

The state of the score reader associated with these actions can be thought of as being closely related to a current position or point of interest in the score. The main significance of this scanning is that it provides a mechanism through which access is gained to the contents of the score.

A common artefact used on two dimensional data structures in computer science, called an iterator, can be used to allow one to access the internals of a data structure.[83] A similar object can be used with a computer model of a score. A score iterator is an object that points into a score. It points to a particular object in the score at any one time. It implements something akin to the current position of Kassler's MIR system, or to the point of interest of MUSIKUS. The iterator may be used to model the act of reading a score. This act may be broken down into the act of reading of individual entities within the score. From the point of view of the reader of the score, various entities within the score may be treated as being read one by one.[84]

The score iterator is an object that points to entities within the score.[85] A score iterator is implemented which points to a single entity in the score at any one time. It has the capability of being located at any entity in the score. Also it can be moved about from one entity to another and/or from one point in time to another in one of a number of general ways. In the case where the entity pointed to is a note or a rest, both of which occupy time, the score iterator is capable of pointing to a time within the entity. Hence if a score iterator points to a note, it may be made to point to the start of a note. This will be the normal case. It may also be made to point to any time within the duration of the note as well. One such internal point might correspond to the point of resolution of a suspension.

---

[83] Borland Borland C++ Library Reference Version 4.0 (Scott's Valley, California 1993), pp.355-462.

[84] It is possible that a human score reader has a capability for observing groups of symbols rather than a single symbol at a time. A chord may possibly be read as a unit, for example. This mode of reading could be simulated in the current implementation, by designing and implementing a special score iterator.

[85] The assembler-like programming language in Michael Kassler, opus.cit., seems to be the first music analysis software system that supported the idea of a current note. MIR was developed in the Department of Music, Princeton University on an IBM-7094 computer.

It is possible to have any number of score iterators of varying types for a single score.

Two basic iterators are presented in **scoreView**. One iterator, a single stave iterator, specialises in visiting entities on a single stave of the score. The other iterator, a multi-stave iterator, specialises in a standard traversal of the entire score. Each of these iterators can operate in MONO or POLY mode. If the score has staves with multiple simultaneous notes, then MONO mode limits traversal to the highest-most notes on one or more staves. POLY mode involves traversing all the notes on one or more staves.

Objects of class ScoreIterator may be declared for a score **s** as follows

```
Score s(filename);
ScoreIterator si(s);
```

If the score **s** has only one stave, a score iterator in MONO mode is created, by default. If the score **s** is a polyphonic score then the score iterator which is created as a result of the above declaration will be a polyphonic score iterator. This iterator gives a standard traversal of the score in POLY mode. If it is required to traverse a single stave of a polyphonic score, a declaration such as the following will create an appropriate score iterator, identified by the name **si0** in this case.

```
ScoreIterator si0(s, 0);
```

The staves in a score are numbered in sequence starting at 0. This score iterator will be in MONO mode by default and will scan the first stave, that is stave number 0, of the score. The mode of a score iterator can be changed by calling the member function

```
si.setScanMode(POLY);
    or
si.setScanMode(MONO);
```

5.9 Locating.

The starting point for access to a score is normally at the beginning of the score, but may also be at some intermediate position. Starting at the beginning and moving to the

right provides all the contextual information required to read the score. In this way, key and time signatures and other score markers are encountered in their proper sequence. Starting at some intermediate point inside the score on the other hand, involves some backward scanning to establish a context in which to read the score. For example, to discover the clef and key signature in a physical score, it is necessary to scan backwards from the score iterator to the last clef and key signature. Such will, at most, involve backward scanning to the leftmost part of the page that is a feature of the physical score and not of the abstract version of the score. For a reader scanning the physical score by beginning from a point other than the start, the time signature might be deduced by inspection of the bar length, possibly with the help of observations on the beaming structure. Alternately this could be done by backward scanning to the last time signature. In a computer implementation, it is highly desirable that such scoping information be resolved automatically by the software, if the programmer-analyst is to be freed from such activities. Of course, in any computer implementation this automatic backward scanning must be done with a view to efficiency as well as transparency.

Using the same objects as in the last section, examples of the use of member functions of the ScoreIterator object, **si,** for positioning it at a specific entity in a score include

```
si.locate();                    // moves si at the start of score
si.locate(NOTE);                // moves si at the first note of score
si.locate(BARLINE, 20);         // moves si to the 20th barline
```

The score iterator object may be made to point to other objects, which may have some kind of adjacency relationship with its current position by means of

```
si.step();                      // moves si to next entity
si.step(NOTE)                   // moves si to the next NOTE entity
si.step(Rat(1,8));              // moves si forward one quaver or
                                // eight note
si.stepb();                     // moves si back to the previous
                                // entity
si.stepb(NOTE);                 // moves si back to the previous
                                // note
```

Which entities are selected as being 'next' depend on the traversal order, which is dealt with in the next section. Functions of the kind used above can be combined. For example, moving the score iterator, **si**, to the first object at the middle position of the 11th bar of a score in 4/4 time, is achieved by

```
locate(BAR, 11);        // locate the start of the 11th bar
step( Rat(2,4));        // move forward a distance of 2/4
```

The entity located by a durational step function, such as **step(Rat(2,4))**, is always the first written entity at that time score time. The end of a note or rest is never a candidate for selection in these cases. One is always guaranteed to have a score entity after such an operation, except in cases where the **step** function causes the iterator to move off the end of the score.

5.10 Traversing.

Having established all relevant contexts, the score reader will normally start at the beginning and proceed to read the score from right to left for reading lines, or to scan up and down, or perhaps in some zig-zag fashion, for harmonies, or in a combination of the two for full score reading. All of this can be broken down into the activities of locating and/or stepping and reading basic entities. In addition to the normal activities of score reading, it is not unreasonable that the human score reader might want to read lines in a score backwards, or in any other possible sequence of accesses. In all cases the reading of the score involves reading basic entities of the score.

A multi-stave iterator in POLY mode follows a path that is described as a standard traversal of the score. The basic principle of this standard traversal is laid out in the following algorithms. Prior to giving the rules, it is useful to define three types of entity. The first type, which includes all entities that have duration, will be referred to as of type A. These consist of notes and rests. The second type, type B consists of any barline, and entities of type C consists of any entity other than those of type A or B, such as clefs or key signatures. Here 'visiting' an object is interpreted as moving an iterator to the object and optionally doing some kind of user-specified processing on it. For each line in a score, the iterator maintains the position of the last entity visited.

A standard traversal is described in the following algorithm. Score iterators of the type we are dealing with, have a current position associated with them on each stave. The iterator's current position corresponds to an entity on a specific stave and also to a point in score time. In a multistave environment, it is useful to think of each stave as having a current position associated with it, which we will call a stave current position to distinguish it from an iterator current position. The system maintains all stave current positions at the same points in absolute score time. When a score iterator moves downwards to the next stave, it does not move forward in time, but to the relevant stave current position. The movement of the iterator current position is either from left to right or vertically downwards in the score. All stave current positions are set to the first entity in the score before the start of the algorithm. The following algorithm in Fig.5.5 gives the rules for any traversal step.

An assumption that is made in this case is that we are dealing with scores where all staves share the same time signature and bar structure. The internals of **scoreView** do not require this to be the case, but the current implementation of a polyphonic score iterator does. A simple change to the algorithm, by classifying barlines as being of type C, would remove this restriction.

Condition	**Action**
At start	Visit first entity on first stave.
Last entity of score on last stave.	Traversal complete - exit algorithm.
Next candidate entity of type C or grace note.	Move the *iterator current* position to next entity on the same stave.
Current entity is of type A with a further type A entity vertically contiguous underneath it on the same stave.	Move the *iterator current position* to the vertically contiguous entity underneath on the same stave.
Last entity or part of an entity processed on the last stave.	Calculate the time slice as the minimum of durations of entities at stave current positions on all staves. Advance all stave current positions by the time slice. Make the uppermost entity on the top stave the *iterator current position*.
Type B entity encountered.	If not on uppermost stave report an error. Visit all the *stave current positions.* If all *stave current position*s are not at barlines, report an error. Step current positions on all staves to next entity. Make *stave current entity* on top barline into the *iterator current* entity.
Any entity.	Move to current position on next *stave current position.*

Fig.5.5 Algorithm for standard traversal.

Comment.

The above algorithm produces a traversal as shown in Figs 5.6, 5.7 and 5.8. Exactly the same principles apply to scores that have multiple notes/rests on the same stave. Note that this algorithm is compatible with traversing both a single stave and multiple stave scores. The standard multi-stave iterator visits all objects in the score in a reasonably natural order. This iterator may itself be used as a base class for the construction of new iterators. It will not be clear, until extensive work is done using **scoreView** what additional iterators might be useful for polyphonic music. In appendix table A3.1, an example is given of the use of this multi-stave iterator to traverse a polyphonic score. The score consists of a section from the start of the sixth movement of Beethoven's string quartet in C# minor, op. 131. The same score iterator is used in the computer to play this score. Grace notes require special treatment. In this implementation of **scoreView** they are treated as if they do not consume any time.

All grace notes on the same stave that are either vertically or horizontally contiguous are visited as a special case of standard traversal before any following entities are processed. Cases that may arise with multiple polyphonic grace notes will require some further attention.

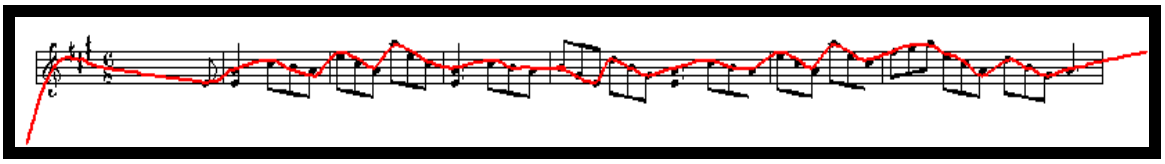


Fig.5.6 Single stave traversal in MONO mode.

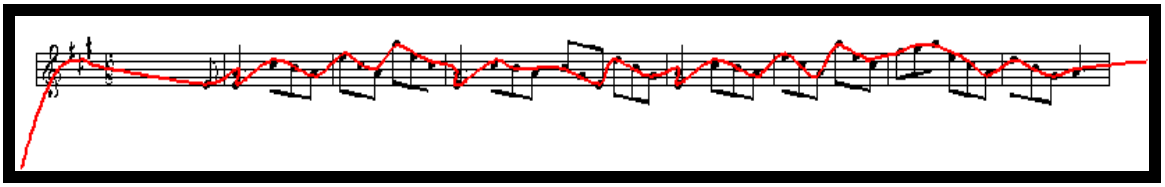


Fig.5.7 Single stave traversal in POLY mode.

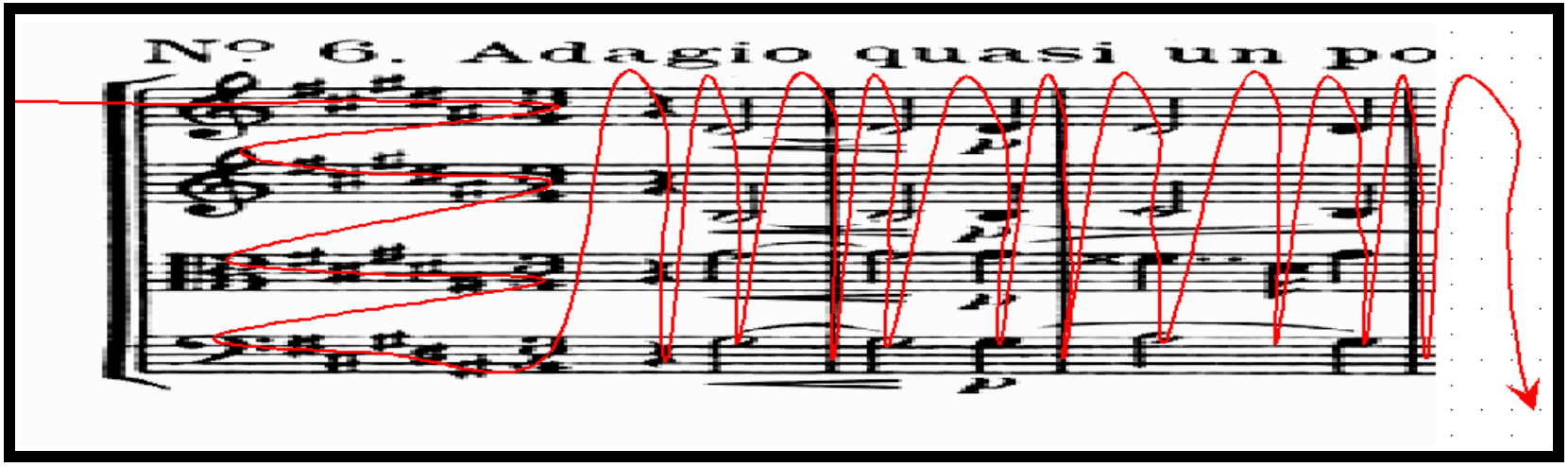


Fig.5.8 Multi-stave traversal in POLY mode.

Elongated image of the start of the sixth movement of Beethoven's string quartet op.131.

In the following two sections the material covered to date will be used to construct some very simple algorithms for performing actions on scores. These are intended as an introduction to the more musically meaningful algorithms of chapters 7 and 8.

5.11 Algorithm 1.

The task here is to examine a monophonic score that is held in a file called "SCORE.ALM" and to output a message that tells us whether the note immediately following the first barline is any 'D'.

```
(1)        Score  s("SCORE.ALM");
(2)        ScoreIterator  si(s);
(3)        si.locate(BARLINE);
(4)        si.step();
(5)        if ( si.isA(NOTE))
           {
(6)         if (si.getAlpha() == "D")
(7)              cout << " Score starts with a D ";
(8)         else cout << " Score does not start with a D";
           }
```

Fig.5.9 Algorithm 1 to identify if the note 'D' follows the first barline.

When this program is run, line (1) causes a model of the score, or a score object to be built in computer memory from the contents of the file "**SCORE.ALM**". The name **s** is associated with this score object. In order to look at the note which is at the start of the first bar, we must have a way of accessing information within the score, of looking into the score, so to speak. A score iterator is used for this. When created, the score iterator is 'looking' at the first object in the score, possibly a clef. Line (2) creates such a score iterator object, called **si** for the score **s**. Line 3 causes the iterator to 'look at' the first barline in the score. Line (4) causes the iterator to step to the next entity in the score. Here we have a problem with interpreting what is required. The original specification used for the algorithm was incomplete as it did not tell us what to do if there is not a note after the first bar line, as in the case, for example, where the first object after the barline is a rest. This is an example of lack of definiteness, one of the basic properties of an algorithm, and is a fault of our original specification of the algorithm. The algorithm must be re-specified so as to rectify the defect.

5.12 Algorithm 1a.

Task is to examine the score that is held in a file called "SCORE.ALM" and to display the following messages

- "Score starts with any D" if the first barline is followed immediately by the note 'D',
- "Score does not start with any D" if the first barline is followed immediately by a note other than 'D'

output nothing otherwise.

Although algorithm 1a has solved one of the problems with algorithm 1, there is still a need to be sure of what the algorithm means.[86]

In the above implementation, the program simply does nothing when a rest is present instead of a note, since line (5) checks that the type of entity being dealt with is a note, before line (6) checks if the note is any 'D' and, depending on the outcome of this test executes either line (7), if a note 'D' is found, or line (8) if a note other than 'D' is found. The input to this algorithm is the score held in the file "**SCORE.ALM**". The output consists of the message displayed on the computer screen.

Writing a program like this is pointless, as we could have answered the question by consulting the score. One case in which automatic analysis becomes useful to a musicologist is when dealing with large corpora. The next program may be used to do the same kind of processing on an unlimited number of scores. It calculates the percentage of scores that start with any note 'D' (algorithm 2). The program uses three variables that appear in lines (1) and (2). **fileName** is used to hold the names of the files containing scores. **countAll** and **countDs** are used for the calculations. The function **getNextScoreNames** reads a file called **NAMES** that contains a list of the scores for processing. On its first invocation the variable **str** is set to the first score in the list. On its second invocation **str** is set to the second score in the list, and so on until all scores are processed. This function returns the value **TRUE** if a score was found, and **FALSE** otherwise, and hence can be used to control the **while** statement.

---

[86] Thus the output of the algorithm could be misinterpreted as saying something about the note at the start of the first bar in the score. This results from confusing bars with barlines. For many scores the note at the start of the second bar will be found as a result of this query. Also we did not check if the note ‘D’ was altered in pitch by the key signature or otherwise. We have interpreted “any D” to include D double flat, D flat, D natural, D sharp and D double sharp.

```
   {
(1)  String str;
(2)  int countAll = 0,  countDs = 0;
(3)  while( getNextScoreName("NAMES", str))
     {
(4)   Score s(str);
(5)   ScoreIterator si(s);
(6)   countAll++;
(7)   si.step(NOTE);
(8)   if ( si.getAlpha()=="D") countDs++;
     }
(9)  cout << "Percentage of D's is " << (countDs * 100)/countAll;
    }
```

Fig.5.10 Algorithm 2 to calculate the percentage of tunes that start on a note of pitch class 'D'.

5.13 Abstraction.

It is of the greatest importance in designing the system, that we do not require the music analyst to carry undue learning or conceptual overheads in the programming environment. In other words we must provide the user with a suitable score abstraction. The approach used here is to provide a score representation cast as an object with a clear user interface. The object-oriented paradigm is used here to represent a score, and also for the entities that constitute a score, e.g. notes, rests, barlines, clefs, time signatures, key signatures, etc. A set of basic member functions and operators are provided which are adequate for present and envisaged future needs.

The principles of abstraction are demonstrated in the above mini-examples that, at no stage, deal with the representation issues of a music score. In order to write a program to process a score we need to create the relevant objects, such as the objects '**s**' and '**si**' above. We also need to know how functions like '**getNextScoreName**', '**locate**', '**step**', '**getTag**' and '**getAlpha**' work. No internal details of score representation were revealed. The only data that was created was precisely that needed to do the job. Integer variables are used for counting, and a character string is used for the name of the file that was being retrieved. These are part and parcel of the task of imperative programming. On the other hand these score constructs are examples of a process of abstraction. These abstractions allow us to perform complete operations by using concepts such as '**Score**' and '**ScoreIterator**', and relieve us of the non-productive task of dealing with the internal complexities of the operations in question.

As alluded to previously, the process of abstraction need not stop here. In using a score representation ourselves, we can build our own abstractions. The score abstraction presented later defines only a basic and near minimum set of operations that can be carried out on a score. From these building blocks, we can build more complex edifices. For example, although the system can represent polyphonic scores, it does not have, as part of its essential structure, any notion of harmony. However, the system can be used to provide building blocks for a 'harmony abstraction'. This process of building a hierarchy of abstractions, is practically limitless, and gives the potential for using the score abstraction to build any conceivable system to an arbitrary level of complexity. By means of a divide-and-conquer strategy, complex systems can be built. Hence problems that seem unmanageable and complex might be made tractable if a way can be found to successively decompose them into components that are simple enough to be expressed algorithmically. The structures are not limited to being a hierarchy. One could, for example, visualise the score representation forming one component in a chain of processing that models the activities of performing and listening to music, where separate systems are built to model a human performer and to model a human listener.

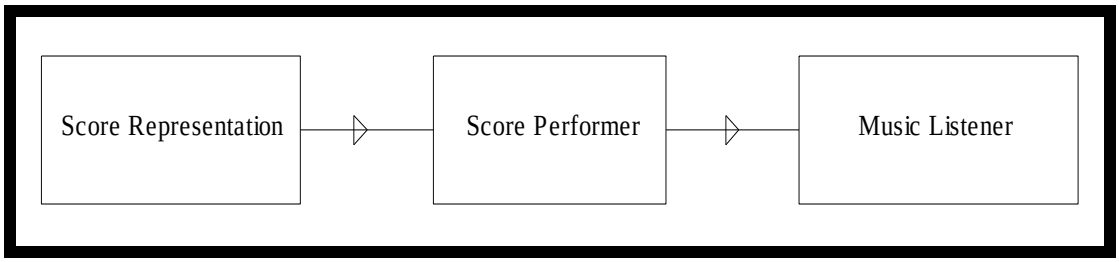


Fig.5.11 Processing in Cognitive Modelling.

The first component here represents a Score model. The second one is a model of a performer who models the playing of a score and the third component models some aspect of human cognition.

Chapter 6. Applications - Verification of Hypotheses.

6.1 Introduction.

A number of applications are given here which use **scoreView** to build programs to do music analysis. These applications have been selected to demonstrate how the system might be used to enable musicologists check the validity of statements made about music. All of the applications given here can be programmed, tested, and run in a matter of a few hours by a competent programmer with a knowledge of **scoreView**. In most cases it took less than an hour to develop a basic version of each program. For clarity, certain simple parts of programs are omitted from most illustrations. Excluded from are some initial details, such as 'include' statements and some declarations, as well as most sections that deal with output. Code that deals with incorrect input has also been removed from the illustrations.

6.2 Structure of Verification.

The bedrock on which any verification is made is the corpus. In the current study, the corpus of music consists of 365 double jig tunes transcribed from "The Dance Music of Ireland" by Frances O'Neill[87] and 54 double jig tunes from "Ceol Rince na hEireann" by Breandán Breathnach[88]. Henceforth these two collections will be referred to as O'Neill's and Breathnach's, or alternately as TDMOI and CRNH1 respectively.

Computer-based verification of a musicologist's hypothesis involves the construction of an algorithm to process the music information in the corpus in such a way as to produce a result that may verify or contradict the original hypothesis. The construction of such an algorithm is not always straightforward. In order to structure this process, it is useful to break the task into a number of steps. Eight steps are proposed here, as one approach to this structuring. These steps are considered under the following headings.

1. Musicologist's text.
2. Related hypothesis.

---

87 Capt Frances O'Neill The Dance Music of Ireland (Chicago 1907).

88 Breandán Breathnach Ceol Rince na hEireann (Baile Atha Cliath, 1963).

3. Algorithm.
4. Decision criterion.
5. Construction of software.
6. Testing of software.
7. Results.
8. Conclusions.

6.2.1 Musicologist's Text.

The first task is to identify statements in the musicologist's text that are suitable for verification. One kind of statement that a musicologist might make is

all X-types have property Y.

Assuming that both 'type' categories and 'properties' in this statement are sufficiently well defined to be identifiable, then we can proceed to the next stage, where we specify an algorithm to test the hypothesis. In the case of the above natural language statement, it is sufficiently well structured to act as the related hypothesis, and here steps 1 and 2 of our structuring process coincide. The selection of text is intimately bound up with the decision criterion of step 4. The decision criterion for the above simple case is easy to formulate. By finding just one exception, that is by finding an X-type with an absent Y-property, we succeed in proving the statement false. If we cannot find such an instance, then we can conclude that the hypothesis is verified by our corpus. This latter case does not exclude the possibility that the hypothesis might be falsified in future. We might, for example increase the number of pieces of music in our corpus, and thereby find exceptions. In the case where a few exceptions to the hypothesis are found, they will inevitably deserve scrutiny. Such exceptions might be found to be misclassified or erroneous items that should not have been admitted to the original corpus in the first case. If a small number of exceptions persist, the musicologist might modify the assertion to something like

normally X-types have property Y.
or in rare cases property Y is absent from X-types.
or usually X-type have property Y.

Here the quantitative nature of the assertion is much less clear. Words such as “normally” do not always carry the same quantitative implications to different people. If a musicologist is claiming that a feature is “usual” in a piece of music, and the purpose of the experiment is to establish the validity or falseness of the claim, it is difficult to pin this word down to an exact percentage that is usable for the decision criterion. If we assume that everyone is in agreement with a claim that a feature of a

pieces of music from a genre is “usual” if it occurs in 95% of cases, then an experiment that verifies this could be said to support the hypothesis. Similarly, if the feature occurs in only 40% of cases, then the experiment could be said to falsify the claim of 'usualness'. It is not at all clear where, in the intervening percentage occurrences, the dividing line between what is “usual” and not “usual” might be. 50%, 55%, 60%, 70%, 80%, 90%? In order to make progress here, two basic questions might be asked. The first question concerns what the author meant to convey. One might be able to interview the author with a view to getting a more quantified version of what was intended. There is, of course, no guarantee that the author would be able to quantify a claim as a percentage, and may retreat to leaving the claim intentionally vague. A second, and a more basic question could be directed at the target audience, to see what the author has succeeded in communicating. It would be perfectly feasible to take a representative sample of readers (the consumers), or potential readers (the potential consumers) of the article and to examine reader responses to the use of such words. Thus for a statement about the “usualness” of a certain feature, one could sample the set of readers of the statement to establish, in quantitative terms, what the statement actually conveyed. Armed with the results of the survey, one could draw on the techniques of statistical sampling theory to quantify what the author's claim conveyed. This would enable the tester of the hypothesis to restate the original hypothesis in quantitative terms, and to proceed with an experiment to validate the statement. This kind of activity would, however, be of use only to a researcher who wished to go to extremes to verify or falsify the results of previous researchers.

An alternate approach, and the one that is adopted in this study, is to avoid making any strong claim about such 'fuzzy' adverbs in advance, by categorising the results as follows -

1) support the original hypothesis, using a conservatively 'safe' criterion.
2) contradict the original hypothesis, using a conservatively 'safe' criterion.
3) for results other than (1) or (2), modify the hypothesis.

6.2.2 Related Hypothesis.

If the musicologist’s text contains statement of the type “all X-types have property Y”, where the “type” and “property” are unambiguous, then as we have seen, the related hypothesis and the original are identical. Very often the hypothesis may need to be extracted from its context and stated afresh. For example, the hypothesis may have to be fished out of more than one sentence of the author. If the hypothesis uses

fuzzy words such as “normally”, it may be useful to specify a schema, giving possible conclusions that we may draw from various hypothetical results. One example of such a schema is given below

Hypothesis: Normally X-types have property Y.
Experiment: Examine all X-types in our corpus and measure the percentage of those with property Y.

Schema:
If percentage is 90 or more, accept the hypothesis.
If percentage is less than 50, reject the hypothesis.
Otherwise modify the hypothesis by including a statement of the percentage.

6.2.3 Algorithm.

The construction of an algorithm involves a formalisation of the hypothesis. What is meant by “formalisation” here is arriving at an unambiguous interpretation of the natural language statement made in the musicology domain for which experimental evidence is being sought, to the level of detail where a program can be written to prove or to disprove or to help to refine the hypothesis. One perspective on the above process is important to make here. We bring our considerable cognitive powers to bear in interpreting natural language statements, which always appear within a context. Much of this happens at a subconscious level. Our software environment does not come equipped with such domain knowledge. Hence the burden lies with the algorithm designer to specify unambiguously and completely what the computer is to do. In this study, we will use natural language to express these formalisms. Natural language is so laden with contextual meanings, that constant vigilance is necessary to ensure that these algorithms involve clear thinking. From the natural language expression of the algorithm, a computer program is constructed.

6.2.4 Decision Criterion.

As discussed in section 6.2.2, the decision rules take the form:

if the percentage is 'x' or more, accept the hypothesis,
if the percentage is less than 'y', reject the hypothesis (where y <= x),
otherwise modify the hypothesis by including statement of the percentage,
where the limits 'x' and 'y' for accepting or rejecting the hypothesis are picked conservatively.

6.2.5 Construction of Software.

This is the stage at which the transition from the algorithm, expressed in natural language, into a formal representation as a computer program is made. We may well discover at this stage that the formalisation process of 6.2.3 may not be a once-off task. It may be necessary to return to step 6.2.3, if incompleteness or other inadequacies are discovered in the formalisation. The end result of this stage is the construction of a computer program, the output of which enables us to test the hypothesis.

6.2.6 Testing of Software.

Following the construction of software, it is essential to check that our program actually does the analysis that is intended. Some of the techniques of software engineering[89] for program testing and verification are desirable. One approach might be to select a small but varied subset of the corpus and to run the automatic analysis on it and then to replicate the process manually, thus enabling cross checking between the manual results and the output of the computer analysis.

6.2.7 Results.

Results consist of the computer output.

6.2.8 Conclusions.

Stating the conclusion involves applying the decision criterion from the output of the algorithm. Typically this is followed by a discussion.

6.3 The Corpus.

The cases presented in this chapter are drawn from a number of statements or hypotheses, made about Irish folk dance music in general. These statements are used for the construction of formal hypothesis at a level suitable for implementation in software. This software is then run using a particular corpus of Irish folk dance music, which forms the evidence against which the claims are tested. Obviously, selection of a corpus is crucial to validifying statements made about a music genre. In this study we are dealing with folk music, which is orally transmitted and from which instances of tunes have been transcribed from musicians by a collector. One may raise here questions about the intent of the transcribers, the organisation of the

[89] G.J. Myers The Art of Software Testing (New York: John Wiley 1979).

publications, the accuracy of the music as a historical record, and about the representative nature of the material in the collection.

On the question of general intent, it is beyond doubt that the collectors were motivated by a desire to preserve and to protect the tradition. Both collectors were performing musicians in their own right. O'Neill was a fiddler and Breathnach an uillean piper. The transcriptions can be perceived as having two different intents. One is the descriptive intent, which attempts to provide a most detailed record of a music performance. The second one is the prescriptive intent whose aim is to provide enough detail to enable a musician to provide a performance[90]. In both collections, the notation is closer to the prescriptive manner, in which most of the micro-details are omitted, with the exception of some grace notes and specific ornaments, such as various rolls and crans. O'Neill occasionally tried to capture some of the rhythmic complexity, using the highly inadequate binary divisions of staff notation. This feature appears in a few tunes in the parent volume[91] from which "The Dance Music of Ireland" was created, but some of these were abandoned in the later publication.

Both publications organise their material into categories of "double jigs", "single jigs", "slip jigs" and "reels", with, in the case of O'Neills the additional category of "long dances, etc" and "miscellaneous". The Breathnach collection was compiled with the help of a thematic index. Consequently it has no duplication of tunes. O'Neill, on the other hand, inadvertently replicated a number of tunes. This is not surprising in a collection of such size that was completed without the use of a thematic index. Breathnach's collection has a wealth of detail about the contributing performers, the instruments used, and about related tunes from other collections. However, for a substantial number of tunes, O'Neill also supplied an associated wealth of detail that parallels the Breathnach collection. Although such detail was omitted from O'Neill's "The Dance Music of Ireland", some of it was included in the parent volume, "The Music of Ireland", which was published in 1903. The parent volume contained 1,850 pieces including 1,100 dance tunes. The later collection "The Dance Music of Ireland" was compiled mainly from tunes printed in 1903, with some additional material included. In "Music of Ireland" a sizeable portion of the tunes bear the name of the musician from whom the music was transcribed. Additional bibliographical information is available about many of the musicians in

---

[90] Charles Seeger "Prescriptive and Descriptive Music Writing" Musical Quarterly, volume 44 (1958), pp.184-195.

[91] Capt. Frances O'Neill's The Music of Ireland (Chicago 1903).

two books written by O'Neill[92]. A small number of Edison cylinder recordings of the musicians who contributed to the O'Neill collection have survived from the period.

On the thorny question of accuracy, we would need access to recordings of all the original musicians, at the original transcription sessions to verify the material. However, in the absence of such evidence, with the exception of a small number of surviving Edison wax cylinder recordings by performers who contributed to the O'Neill collection, and surviving tapes by contributors to the Breathnach collection, we can glean indirect evidence of the general accuracy of the collections. The Breathnach collection was first published a little over thirty years ago and was, in the main transcribed from tape recordings of contemporary folk musicians, many of whom have died in the intervening years. Many of these musicians were instrumental in shaping the style of the current generation of musicians and acted as exemplars for these young people. The Breathnach book was used by such young learners, many of whom had access to Breathnach's informants as well. The lack of criticism of the book provides indirect testimony to its overall accuracy. In the case of the O'Neill collection, one may have a few more reservations. The consistency of the notation is not as good as in the Breathnach collection, and it contains a small, but significant portion of errors. However the popularity of the O'Neill publications, in what was a predominantly oral tradition, in which a minority of musicians could read music, combined with the relative absence of any substantial criticism of the book by traditional musicians attest to the validity of its material. The "Dance Music of Ireland" acted as a standard reference to such an extent that musicians frequently referred to it as “The Book”. The popularity of the O’Neill collection can be gleaned from the number of re-issues or re-edited versions of it that have been produced. This contrasts with the fate of reissues of some of the earlier collections, notably the Joyce and Petrie publications, which did not gain such widespread acceptability.

Breathnach[93] gives us an account of how the work of transcription was made for the O'Neill publications.

> " .. and the task of notation was undertaken systematically. Tunes were noted down by James O'Neill from the playing, singing, whistling, lilting. and even the humming of contributors, played back, and corrected or accepted as the case might be."

---

92 Capt. Frances O'Neill Irish Folk Music; A fascinating study (Chicago 1910) and Capt. Frances O'Neill Irish Minstrels and Musicians (Chicago 1913).

93 Breandán Breathnach Folk Music and Dances of Ireland (Cork: Mercier Press, Revised Edition 1977), pp116-117. Breathnach's source of evidence here is not given. The most likely source is from O’Neill’s contemporaries.

The most telling point about the O'Neill collection is that musicians by and large feel comfortable with it. The music in it is close enough to musicians' expectations to merit acceptance. An additional point in favour of acceptance of the material arises from the fact that O'Neill and his transcriber were performing musicians, steeped in the tradition. Even if an occasional mis-transcribed note got into some tunes in the publication, the end product would, most likely have been filtered by the compiler or by his transcriber, and consequently would have been acceptable to them. In effect, they would have acted as a filter that might be expected to reject invalid syntax of the style current at the turn of this century. This last consideration is crucial in cases where some material was included in the O'Neill collection that was drawn from earlier written sources.[94]

Assuming that we are in the business of verifying hypotheses about the current living tradition, the question arises about the admissibility of these collections as **representatives** for the purpose of verifying statements about the current living tradition. The Breathnach collection is now over thirty years old. The main point in favour of the validity of use of the Breathnach collection lies in the fact that many of the contributors to the Breathnach collection were exemplars for the current generation of musicians. With the O'Neill collection, which was made over 90 years ago, we are on shakier ground. A temporal span of 60 years exists between it and the Breathnach collection. However, the folk tradition during these spans of time was an inherently conservative one, which changed at a very slow rate. No significant new genres emerged over the last 100 years. One source of conservatism reflects itself in the way in which practising musicians hold key older players as exemplars to be copied. Additionally, general acceptability by musicians of the O'Neill book, suggests that it consists of valid representations of the tradition. For the purpose of the current study, both the Breathnach and the O'Neill collections will be accepted as valid corpora in support of proofs. However separate analyses will be carried out on both collections with a view to exercising caution by being vigilant to differences.

---

[94] In his introduction to The Dance Music of Ireland, O'Neill give some instances. "Denis Delaney (No.7) is a good specimen of an Irish jig with three parts, forgotten in Ireland, yet preserved in "Crosby's Irish Musical Repository," published in London in the year 1810. Numbers 168, 190 and 198 were found in the extremely rare "Repository of Scots and Irish Music, " printed in Edinburgh in 1799. Number 982 was found in the volume of country dances of 1798 before mentioned, while numbers 254, 355, 356 and 357 were discovered in "The Hibernian Muse," published in the year 1797." O'Neill also mentions in his introduction, the inclusion of two tunes from a manuscript by Mr. Timothy Downing.

A further issue concerns the representative nature of the corpora and their validity for verifying statements about Irish dance music in general. The corpora used consist of double jigs alone, which represent only one genre within the tradition. It is, however, a major genre, the other main ones include reels, hornpipes, single jigs and slip jigs. Jigs form the second largest genre in both collections. There are 365 double jigs out of a total of 1001 tunes in O'Neill's and 54 out of 214 in Breathnach's. In both collections the most frequent genre is the reel, which is of more recent origin.[95] Any general, unqualified statement about Irish dance music could be expected to apply to all genres, including the double jig.

Yet another issue concerns the accuracy and completeness of the computer representation of the corpus. In the current study, the computer version of the corpus was created initially as text files in ALMA code. Additional checks on the accuracy of each entry were made aurally, by checking the original printed source against a computer performance using a MIDI synthesiser. A further visual check was carried out by comparing the original printed source against a computer generated printed output.[96]

6.4 The Text.

The following is an extract from “The Creative Process in Irish Traditional Dance Music”.[97] The author prefaces this extract by focusing on attempts to look at systems of performance technique and tackling issues such as improvisation and the creative process in general, and declares that the paragraphs represent an attempt to view the tradition from within . "Any insights in this paper are offered in the same spirit, in that they stem directly from the subjective experience of performing traditional dance music over the past two decades"

---

[95] Breadnan Breathnach, op.cit., 1989, p. 137.

[96] A program A2S.CPP to generate a text input file from **scoreView** was developed for use with the well-known 'SCORE' printing program of Leland Smith.

[97] Dr. Micheál Ó Suilleabháin “The Creative Process in Irish Traditional Dance Music” Gerard Gillan and Harry White Irish Musical Studies (Dublin: Irish Academic Press 1990), pp. 117-130.

THE FRAMEWORK

Irish traditional dance music, with very few exceptions, is constructed from basic eight-bar units which, in the standard piece, are made up as follows:

8 bars	8 bars	8 bars	8 bars
A	A	B	B

These units form an important part of the conceptualisation of the musicians and are referred to as 'parts'. The concept of bar has little or no significance as far as the musicians are concerned and is only used by those who are musically literate (still a minority) and, even then, mostly in the context of notation. If such is the case, how do traditional musicians sense the unit in question? In my opinion, the part is perceived on two interactive levels - through the feeling of eight main rhythmic pulses, and through the melodic framework pointed by tonal cadences.

The standard dance-piece consists of two different parts, each being 'doubled' (i.e. repeated with or without some modification). Occasionally in certain pieces, the parts are 'singled' (i.e. not repeated) but this is very much the exception and applies mainly to reels. Furthermore, in a relatively small number of cases, dance-pieces are found with three or more parts. Regardless of the number of parts, however, by playing through the entire piece once, the musician covers what he calls one 'round' A round, therefore, may be thirty-two bars in length (if the parts are doubled in standard two-part piece) or sixteen bars (if the parts are singled). Similarly, the length of the round increases with the addition of extra parts which normally reach a total of no more than six.

In a standard two-part piece with the parts doubled, the thirty-two bar round is perceived by the musicians as being in two distinct sections: the first half (covering the first part and its doubling) which is called the 'tune', and the second half (i.e. second part and its doubling) which is called the 'turn'. The tune is normally contained with the lower octave, while the turn usually breaks into the higher octave. There is an ambiguity here in folk terminology in that the word 'tune' can refer to the first half of a round, and also to the piece itself in its entirety.

Fig.6.1 From “The Creative Process in Irish Traditional Dance Music", pp.115-6.

This text is laced with fuzzy words that indicate some of the difficulties that are encountered in providing a simple enough picture to capture the main features without seriously interrupting the main points being made by including too much detail. Relevant words and phrases that are used are given below, with the numbers appearing in brackets indicating repetitions.

exception(s)(2), standard(4), mostly, occasionally, relatively small number of cases, normally(2).

Questions arise about what is conveyed by words like “normally”. Does “normally” correspond to over 99% of cases? If it does, then we could regard the exceptions as being some kind of very special cases that are almost out of the genre proper, or as experimental cases. If, on the other hand, the word “normal” is used to represent 80% of cases, then we are in a very different situation. One could also ask how a precise meaning might be attached to the expression "relatively small number

of cases". Intuitively, this might seem to one as being less than about 5% of cases, but there is no way of knowing that this is what the author intended.

Many of the claims made above relate to the body of Irish dance music as practised by musicians. We can look at printed and manuscript sources to verify the claims made by the author. Conclusions drawn are based on the hypothesis that the printed versions represent an accurate record of the oral tradition in respect of the aspect under test.

Possible sentences for verification include

1. The standard dance-piece consists of two different parts, each being ‘doubled’ (i.e. repeated with or without some modification).

2. Occasionally in certain pieces, the parts are 'singled' (i.e. not repeated) but this is very much the exception and applies mainly to reels.

3. Furthermore, in a relatively small number of cases, dance-pieces are found with three or more parts.

4. .. the length of the round increases with the addition of extra parts which normally reach a total of no more than six.

5. The tune is normally contained with the lower octave, while the turn usually breaks into the higher octave.

Since the current corpus available for automatic processing consists mainly of double jigs, we look for testing of these against double jigs only. No extra programming effort is required to verify the assertions for other dance genres. All is needed is to have a larger and more diverse corpus.

The first step is to arrive at formalised assertions that might be the subject of experimental verification. Statements 1, 2 and 3 serve to say things about the number of parts and whether each part is played once or directly repeated.

6.5 Experiment 1: Singled Versus Doubled.

Ex.1.1 The text: Points no 1 and 2 above, that is:

1. The standard dance-piece consists of two different parts, each being 'doubled' (i.e. repeated with or without some modification).

2. Occasionally in certain pieces, the parts are 'singled' (i.e. not repeated) but this is very much the exception and applies mainly to reels.

Ex.1.2 The Related Hypothesis - Introduction.

The original text contains a number of points that could be verifiable, including that the standard dance-piece consists of two different parts, each being 'doubled', i.e. repeated with or without some modification. The first assertion is that the standard piece consists of two parts. We will defer checking this until later. The second statement gives exception to part of the first rule, but needs some interpretation, as to what the author intended to say.

2. 'Occasionally in certain pieces, the parts are 'singled' (i.e. not repeated) but this is very much the exception and applies mainly to reels.'

It is not clear if the second statement applies to two-part standard pieces only, or to pieces with more than two parts as well. We will assume here that it applies to all pieces.

Some clash of terminology here is inevitable. Jigs fall into the categories of single jigs, in 6/8 time with a predominant crotchet quaver crotchet quaver rhythm, double jigs, in 6/8 time with a predominant 3+3 quaver rhythm, and slip jigs, in 9/8 time with a 3+3+3 quaver rhythm. As these have already been pre-classified by the collector, we need not concern ourselves with the classification problem. A 'singled' double jig is a double jig in which each part is played once per round. A necessary condition for a notated singled double jig is the absence of a repeat sign in bar 8. However the absence of a repeat sign does not guarantee us that it is a 'singled' double jig. The reason for this is illustrated below, where a, b, a1, a2, b1 and b2 represent 8-bar segments of pieces.

8 bar segments in a 'singled' double jig with two parts may be represented by

a b

8 bar segments in a 'doubled' double jig with two parts may be represented by

a a b b

In the manuscript the 'doubled' parts are not written out twice, in cases where they are identical, but are represented as shown symbolically below ('://' represents a repeat sign).

Hence a b gives a 'singled' double jig
a :// b :// gives a 'doubled' double jig.

A problem arises in the case where a 'doubled' double jig has been transcribed from a player who has introduced sufficient variation in the repeated 'a' part to merit writing out both versions of either/or the first and second 8 bar segments".[98]

a1 a2 b ://
or a1 a2 b1 b2

Hence, the notated 'doubled' double jig piece with two parts might be confused with a four-part 'singled' double jig piece. In order to solve this problem, we need a piece of software that will identify whether one 8 bar segment is sufficiently close to another to be regarded as a variant.

The solution to determining whether a piece is 'singled' or 'doubled' proposed here is achieved by implementing the following set of rules or algorithm. The rules are searched in order in which they are given, and the first rule that applies is taken as the answer.

[98] Jig tunes are also notated with alternate endings. For example, a jig tune in which the end of a section is varied, but where the first, say, seven out of the eight bars are the same, the part may be notated with 9 bars of notation, with two alternative endings following the first seven bars, e.g. as in TDMOI nos 1, 71, 90. This method of notation is more frequently employed in the second parts (the turns) of tunes. The current corpus avoids these complications by fully representing such parts. Repeat signs are used only at the end of eight bar segments.

Algorithm

Rule	Action
if the piece has a repeat sign in bar 8	classify as 'doubled'.
else if the piece is exactly 16 bars long	classify as 'singled'.
else if bar 1-8 is similar melodically to bar 9-16	classify as 'doubled'.
else admit all pieces	classify as 'singled'

Fig.6.2 Algorithm for classifying tunes as 'singled' or 'doubled'.

Some comments required about the above rules.[99] As a formal statement of the algorithm, there is one glaring omission. We have not given any formal meaning to what "similar melodically" means. For the present, we will hedge the issue by proposing the existence of a function, called diff1[100], which takes 3 parameters. The function diff1 evaluates the melodic difference between two line segments of music and returns a number that is an estimate of the melodic distance between the two line segments of music. The first two parameters in this function are score iterators that represent the starting positions of the two melodic lines, and the third parameter is the length of the two segments to be compared expressed in rational units.[101]

The musicologist's text uses the word “occasionally” and “very much the exception” and “applies mainly to reels” in reference to the relative frequency of “singling”. This might imply that we should expect to find a small percentage of reels (less than 10%, say) 'singled', and for other categories, such as double jigs, we should find a still smaller percentage 'singled' (5%, say).

[99] An assumption here is that this algorithm deals only with tunes that are at least 16 bars in length. An additional rule to check this would have to be inserted if the algorithm were to be used to check for inadmissible tunes, shorter than 16 bars. For the corpora, this is unnecessary as, in the case of double jigs, all are at least 16 bars in length, and a check of this was performed when the corpus was created. Also a check is carried out during the creation phase of the corpus that all tunes in the double jig section are in multiples of 8 bars.

[100] **diff1** is a simplified implementation of the more general **difference** function which is documented in Appendix 1. Melodic difference is calculated by diff1 on the basis of pitch differences weighted by window durations. Contour, metric and note durations are not processed by **diff1** and transposition processing is not done.

[101] From running this function on the corpus of music, it has been found that a returned value of less than 300 indicates an appropriate measure of melodic closeness.

Ex.1.2 The Related Hypothesis : Statement.

The occurrence of 'singled' double jigs tunes is very rare.

Ex.1.3 Algorithm.

Visit each tune in the corpus.

Apply algorithm 1 to each tune, and count the numbers that are classified as 'singled' and also count the total number of tunes visited.

When all the tunes have been visited, apply the decision criterion below.

Ex.1.4 Decision Criterion.

If the percentage of singled tunes is 5 or less, confirm the hypothesis.

If the percentage of singled tunes is greater than 5 and less than 10, quantify the hypothesis.

Otherwise contradict the hypothesis.

Ex.1.5 Construction of Software.

The coding for classifying a score as 'singled' or 'doubled' is given below. The implementation is cast as a function that returns **TRUE** if the piece is 'singled' and **FALSE** otherwise.

```
int isSingled(Score & s)
{
 // does the score have a repeat sign in bar 8 ?
 ScoreIterator si(s);          // create a score iterator.
 si.locate(BAR,8);             // locate the start of bar 8,
                               // move to the start of bar 9.
 // move to next barline
 si.step(BARLINE);

 // check if barline is one with a repeat sign.
 // (first rule of algorithm).
 if ( si.getBarType() < Set(CLHLC, CLLC, CLH, CLL, CLC, CL))
  return FALSE;

 // next we can check if the tune has more than 16 bars,
 // by searching for bar 18. (second rule of algorithm).
 if ( ! si.locate(BAR, 18)) return TRUE;

 // next we check for melodic similarity by comparing two segments,
 // one starting at the beginning of bar no 1, and the other
 // beginning at the start of bar 9.  The span of the scan is taken
 // here as seven and a half (or 15/2) bars plus an eight note.
 Rat span = si.getTimeSig() * Rat ( 15, 2) + Rat(1,8);
 ScoreIterator si1(s), si2(s);
 si1.locate(BAR, 1);   // position one iterator at bar 1.
 si2.locate(BAR,9);    // position the other at bar 9.

 // calculate the melodic distance between the two segemnts.  A
 // difference of 300 is found to provide a satisfactory dividing
 // line between 'similarity' and 'difference'.
 // ( third rule of algorithm).
 if ( diff1( si1, si2, span ) < 300 ) return FALSE;

 return TRUE;  // last rule of algorithm.
}
```

Fig.6.3 Program of algorithm to verify hypothesis of Ex.1.

Ex.1.6 Testing of Software.

Testing here consisted of printing output on a tune-by-tune basis, for the 54 double jig tunes in the Breathnach collection, and manually checking the accuracy of the results.

Ex.1.7 Results.

```
'singles' analysis on file =\mdb\crnh1\djig.dir

Number of 'singles' is 4 out of 54 (7%)
```

Table 6.1 Output of program of Ex.1 for CRNH1.

```
'singles' analysis on file =\mdb\tdmoi\djig.dir

Number of 'singles' is 2 out of 365 (1%)
```

Table 6.2 Output of program of Ex.1 for TDMOI.

Ex.1.8 Conclusions.

For the combined corpora, we find that (4+2)/(54+365) = 1.4% of the tunes in the corpus are singled, and conclude that the hypothesis stated in Ex.1.2 is supported.

On independently comparing the results from Breathnach's and O'Neill's, there appears some cause for concern. We see that the percentage of 'singled' double jigs is just under 1% for the O'Neill's and is 7% for Breathnach's. One interpretation of this is that the practice of playing jigs in the 'singled' manner has increased in frequency in the time that elapsed between the two collections. The situation may not be as rigid as would be implied by the results. It is possible that tunes may be played as singled or doubled on different occasions. Breathnach writes about this, mainly in relation to reels[102], where he indicates a highly variable practice in relation to reels, but does not mention jigs in the same context. This is a little strange, as a significant proportion (7%) of 'singled' double jigs appears in his own collections.

In the case of this experiment we have succeeded in verifying the hypothesis in Ex.2.1, and at the same time we have raised important questions, that require further investigation.

In this experiment we have made the assumption that singled double jig tunes are always singled in their first parts. From observing the corpus this seems always to be

---

[102] Breandán Breathnach: Ceól agus Rince na hEireann (An Gúm, Baile Átha Cliath 1989), pp. 130-131. 'Faoi dhó a chastar gach cuid sna poit dhúbailte, sna poirt singile agus sna cornphíopaí. Uair amhain a chastar na codanna atá sa phort luascach. Ar cheachtar den dá bhealach a chastar an ríl anois ach fadó, nuair ba le haghaidh damhsa a chastaí í, ní dheantaí na codanna a chasadh ach uair amháin as a cheile. Faoí dhó a chastar gach cuid sna poirt dúbailte, sna poirt shingile and sna cornphiopaí. Is eard a dheantar le ríleanna anois gach cuid a chasadh faoi dhó; ach i gcás ríle nach mbionn aon athrú de bhrí idir an dá mhir sa chuid, ní dheantar an cuid nó an chaoince sin a chasadh ach uair amháin as a cheile. Is de ghrá an líostacht a sheachaint a dhéantar e seo. Fágann sin go bhfuil ríleanna ann a gcasfaí cuid amháin iontu faoi dhó agus cuid eile ionta uair amháin'.

Translation: In double jigs, single jigs and hornpipes each part is doubled. In slip jigs each part is singled. Nowadays, reels may be played either way, but long ago, when played for dancing, each part was played only once. Each part of double jigs, single jigs and hornpipes was played twice. Current practice with reels is that each part is played twice, except in the case of a reel that has similar sections in a part, the tune or the turn is played only once. It is to avoid monotony that this is done. Hence there are reels in which one part is played twice and another played once.

the case. The algorithm would need to be extended if we are to be absolutely sure that no exceptions occur in the form of hybrids that involve the doubling of the first part followed by a singled part.

6.6 A Specialised Class.

It is important, in the analysis of dance music, to have a convenient way of identifying the various parts of a dance tune. Such a facility might be reused repeatedly for building various types of analysis. Hence it is worthwhile to build a piece of software to automate this process. What we need here is a 'parts expert' object for a score, which answers questions such as:

How many distinct parts are played in one round of the piece?
In which bar do we find the start of part **n**?
Does the piece have an odd number of parts?
Is the piece singled? (this is an incorporation into the class of the code above).

The code for this class is given in appendix A2.1. A summary for the public interface of this class is given below.

Constructor:

PartsExpert(Score & s);

Member function to indicate if the piece is singled:

int isSingled();

Member function to return the number of parts in a score:

int numberOfParts();

Member function to return **TRUE** if the piece has an odd number of parts present:

int hasOddPart();

Member function to return the first bar number of part **I**:

int getBarNoForPart(int i);

Note that the bar number **1** is from the first complete bar of the tune. Upbeats are ignored.

6.7 Experiment 2: Number of Parts.

Ex.2.1 The Text.

" Furthermore, in a relatively small number of cases, dance-pieces are found with three or more parts".

Ex.2.2 The Related Hypothesis.

The percentage of tunes with more than two parts is relatively small.

Ex.2.3 Algorithm.

Visit each tune in the corpus.

Apply the **PartsExpert** to each tune and count and record the number of parts in the tune and the total number of tunes processed. Calculate the percentage of tunes with more than 2 parts.

Apply the decision criterion.

Ex.2.4 Decision Criterion.

If the percentage is 15 or less, confirm the hypothesis.

If the percentage is greater than 15 modify the hypothesis by quantifying it.

Ex.2.5 Construction of Software.

We have already taken a large step in this in that we have a function to determine if the piece is singled. Most of the work in calculating the number of parts is done in the parts expert constructor for the class, which counts the number of 8-bar segments. A double section count is made when a repeat sign is encountered. Using the partsExpert class makes this task easy to specify. The entire code consists of the 12 lines in Fig.6.4.

```
while (getNextScoreNames(argv[1], fname))
{
 Score s(fname);

 PartsExpert partsExpert(s);
 countAll++;
 if ( partsExpert.hasOddPart() ) countOddOnes++;
 if ( partsExpert.isSingled() ) countSingles++;

 String sparts;
 sparts.cvtNs(partsExpert.numberOfParts());
 store.put(sparts);
}
```

Fig.6.4 Program to find the number of parts in a dance tune.

Ex.2.6 Testing of Software.

Testing here consisted of printing output on a tune-by-tune basis, for the 54 double jig tunes in the Breathnach collection, and manually checking the accuracy of the results.

Ex.2.7 Results.

The output produced by running this program on the corpus of jig pieces from the two collections given previously is given in tables 6.3 and 6.4.

```
Analysis of Number of Parts in Dance Pieces
taken from file =d:\mdb\crnh1\djig.dir

     Parts        Frequency      Percentage
-------------------------------------------------

       5           1               1
       4           4               7
       3           5               9
       2           44              81
-------------------------------------------------
```

Table 6.3 Analysis of the number of parts in jig tunes from CRNH1.

```
Analysis of Number of Parts in Dance Pieces
taken from file =d:\mdb\tdmoi\djig.dir

     Parts        Frequency      Percentage
-------------------------------------------------
       7           2               0
       6           5               1
       5           4               1
       4           14              3
       3           49              13
       2           290             79
       10          1               0
-------------------------------------------------
```

Table 6.4 Analysis of the number of parts in jig tunes from TDMOI.

Ex.2.8 Conclusions.

We can see here that the number of pieces with two parts is remarkably stable between the two collections, with the older one yielding 79% and the more recent one yielding 81%. The percentage from the combined results is (44+290)/(54+365) = 80%. Hence out the term used 'relatively small proportion' is close to 1 out of five. Our result here leads us to modify the hypothesis to:

Approximately 20% of double jig tunes have more than 2 parts.

One welcome side effect of this analysis is that we can see at a glance, the relative frequency distributions of the number of parts, with between two and ten parts per tune. This confirms the assertion that the number of extra parts normally reaches no more than six. Here we interpret some ambiguity in the statement by assuming that it refers to a total of six, and not to eight. We see that there are no tunes with six parts or more in the Breathnach collection and there are 3 out of 365, or 1% in the O'Neill collection with more than six parts. One tune in the O'Neill collection has ten parts.

6.8 Experiment 3: Ranges of Tune and Turn.

Ex.3.1 The text.

" The tune is normally contained with the lower octave, while the turn usually breaks into the higher octave".

Ex.3.2 The Related Hypothesis.

We already have a clear agreement on what constitutes the 'tune' and 'turn', so we can equate the related hypothesis with the text.

Ex.3.3 Algorithm.

Visit each tune in the corpus.

Scan all notes in the 'tune' part, then extract the chromatic pitch numbers of the highest pitch found, **p1**, and the lowest pitch found, **p2**. Scan all notes in the 'turn' part and extract the chromatic pitch number for the highest pitch found, **p3**. Count the piece as verifying the hypothesis if both of the following conditions are met

p2 - **p1** is less than or equal to 12
p3 - **p1** is greater than 12

Many dance pieces have from one to five notes before the first bar, as an anacrusis that precedes the first accentuated beat. The cumulative duration of these notes is never more than a crotchet in the corpora under study.[103] In pieces that have this phenomenon, the first eight bar phrase stretches from before the first bar through a total distance of exactly 8 bars in duration, and consequently does not reach the end of the 8th bar of the tune. For the present, we will just note that there is a problem here, and defer a decision on how we will handle it in the section titled 'decision 1'. The words 'normally' and 'usually' in the sentence "The tune is normally contained with the lower octave, while the turn usually breaks into the higher octave" need some attention. Do we have a composite hypothesis here about the range of the tune part, in relation to itself, and about the range of the turn, in relation to the range of the tune part, or do we have two separate hypotheses, one about the range of the tune part, and a second one about the range of the turn part? The word “breaks” suggest that we are dealing with a composite hypothesis, and that a statement is being made about a tune which is in the lower octave, with the turn breaking new ground going into the higher octave.

Two points of clarification must be made -

Decision 1: This concerns start and stop points of our scans. In this case we have to choose whether to perform our scan so that it covers eight bars from the first note of the piece, or whether we omit any introductory notes from consideration. From an examination of different manuscripts, one often finds that there exist different versions of the same basic piece in which these notes are omitted. Many cases can be found, for example, by comparing CRNH1 NO 13 with TDMOI NO 24, and CRNH1 20 with TDMOI 158. The decision we will take here is to skip such notes in our calculations and to use only notes from the start of the first full bar of each 8 bar segment, up to and including the note at the centre of the 8th bar. The main justification for this lies in the optional nature of these notes at the start. This enables us to carry out comparisons on a standard form of each tune segment from all the double jigs. The above decision also involves ignoring the notes in the last part of bar 8 also, as these may form an analogous lead in for the second part.

Decision 2: A further refinement concerns a decision on whether to include grace notes in our calculations. This is an example of the kind of snag about which we have to be vigilant. We will ignore grace notes in this case, as they are used for cuts, whose

[103] A study of the melodic structure of the anacrusis is given in 7.2.

purpose is to give rhythmic emphasis rather than to contribute a purely melodic component by their pitch. As grace notes are normally found above the following note, a decision to include grace notes might militate in favour of over-estimating the instances of tunes that violate the first clause of the hypothesis, i.e. that the tune part is contained within the first octave. In relation to the second clause, about the turn part, inclusion of grace notes might lead to an over-estimation of tunes that support the hypothesis.

Ex.3.4 Decision Criterion.

If the percentage of tunes is greater than or equal to 20, accept the hypothesis,
else if the percentage of tunes is less than 50, reject the hypothesis.
else quantify the hypothesis.

Ex.3.5 Construction of Software.

This is a most straightforward application in which a scan is made of the first 8 bars in order to find the maximum and minimum pitches present in each part. A simple comparison of the differences between the maxima and minima can then be used to verify the results. A further scan is made of the 8 bars of the turn, and the maximum pitch is calculated. The calculations in Ex.3.3 are performed for each tune. An annotated version of the main part of the program is given below in Fig.6.5.

```
String fname;
int countTarget = 0;
int countAll = 0;
int part1InOctave = 0;
int part2OutsideOctave = 0;

while (getNextScoreNames(argv[1], fname))
{
 Score s(fname);
 int highestPitchPart1 = 0;
 int lowestPitchPart1 = INT_MAX;
 int highestPitchPart2 = 0;
 int lowestPitchPart2 = INT_MAX;
 ScoreIterator si(s, 0);

 while ((si.getBarNo() <= 8 ||
         ( si.getBarNo() == 8 &&
         si.barDist() <= si.getTimeSig() * Rat(1,2) + Rat(1,8))) &&
        !si.isLast())
 {
  if ( si.getTag()==NOTE && !( GRACE_NOTE < si.getAttributeSet()) )
  {
   if ( highestPitchPart1 < si.getPitch12())
    highestPitchPart1 = si.getPitch12();
   if ( lowestPitchPart1 > si.getPitch12())
    lowestPitchPart1 = si.getPitch12();
  }
  si.step();
 }

 PartsExpert partsExpert(s);
 int nextBar = partsExpert.getBarNoForPart(2);

 while ((si.getBarNo() <= nextBar+7 ||
         ( si.getBarNo() == nextBar+8 &&
         si.barDist() <= si.getTimeSig() * Rat(1,2) + Rat(1,8))) &&
        !si.isLast())
 {
  if ( si.getTag() == NOTE && !( GRACE_NOTE < si.getAttributeSet()))
  {
   if ( highestPitchPart2 < si.getPitch12() )
    highestPitchPart2 = si.getPitch12();
   if ( lowestPitchPart2 > si.getPitch12() )
    lowestPitchPart2 = si.getPitch12();
  }
  si.step();
 }
 int lowestOverallPitch = lowestPitchPart1 > lowestPitchPart2 ?
                     lowestPitchPart2 : lowestPitchPart1;

 if ( highestPitchPart1 - lowestPitchPart1 <= 12 &&
      highestPitchPart2 - lowestPitchPart1 > 12 )  countTarget++;

 if ( highestPitchPart1 - lowestPitchPart1 <=12 ) part1InOctave++;
 if ( highestPitchPart2 - lowestPitchPart2 > 12 )
                                          part2OutsideOctave++;

 countAll++;
}
```

Fig.6.5 Program for testing hypothesis of Ex.3.

Ex.3.6 Testing of Software.

A complete listing of the results for each tune in the CRNH1 corpus was printed out, and the results verified manually against the manuscript.

Ex.3.7 Results.

```
Number of pieces with 1st part in lower octave and
2nd part going into upper octave from =d:\mdb\crnh1\djig.dir
is 15 out of a total of 54(27%)

Number of pieces with 1st part in octave range is 15(27%)

Number of pieces with 2nd part outside octave range is 51(94%)
```

Table 6.5 Output of program for Ex.3 using CRNH1.

```
Number of pieces with 1st part in lower octave and
2nd part going into upper octave from =d:\mdb\tdmoi\djig.dir
is 76 out of a total of 365(20%)

Number of pieces with 1st part in octave range is 100(27%)

Number of pieces with 2nd part outside octave range is 307(84%)
```

Table 6.6 Output of program for Ex.3 using TDMOI.

Ex.3.8 Conclusions.

The average percentage of cases for which the hypothesis is true between both collections is 22%. This is found by a weighted average of the results for CRNH1 at 27% and for TDMOI at 20%. Hence we restate the hypothesis -

The tune part is contained in the lower octave, and the turn part breaks into the higher octave in approximately 22% of cases.

Interestingly, the percentage of pieces that meet both criteria is the same as the percentage that meets the first criterion in the case of CRNH1. A quick visual scan of printed sources indicates that similar results might be expected from the other main form, the reel.

The turn of a piece however, tends to be higher than the first part. One way in which this might be expressed and tested is to assert that, on average, notes in the turn are higher in pitch than notes in the first part. This could be readily tested, by a

simple algorithm. The result of such (see algorithm in appendix A2.2) is given below for the older collection in Table 6.7.

```
Average pitches analysis of parts 1 and 2 of pieces.
Files used from '=d:\mdb\tdmoi\djig.dir'.
Pieces with higher average pitch in 2nd part = 331 out of 365 (90%).
```

Table 6.7 Average pitches for TDMOI.

The average taken here is the unweighted average of the chromatic pitch number of the notes in the first and second parts.

6.9 Experiment 4: Set Accented Tones.

In the same chapter of the previous quotations, Ó Suilleabháin proposes a theory of set accented tones[104] as follows -

> "Within a performance, the musician would appear to be holding on to certain individual tones which occur at important accentuated points. It is the occurrence, or deliberate non-occurrence, of these tones which appears to provide the necessary point of reference for the performer. Illustration 3 shows a typical setting of the opening of the four bars of the double-jig "The Old Grey Goose" (example (a)) with the eight set accented tones boxed. In order to demonstrate that these tones are at the heart of the piece's identity and that any extended interference with them is in the nature of a contradiction of the tune itself, I have included five projected variants of my own . . . . ."

Fig.6.6 From "The Creative Process in Irish Traditional Dance Music", p.123.

[104] In an interview with the author, Professor Micheál Ó Suilleabháin clarified two points about set accented tones. The use of the word set here, means 'fixed'. In effect “set like a jelly”. Also the word tone refers only to pitch and has not a connotation of timbre. (date: 26.11.94).

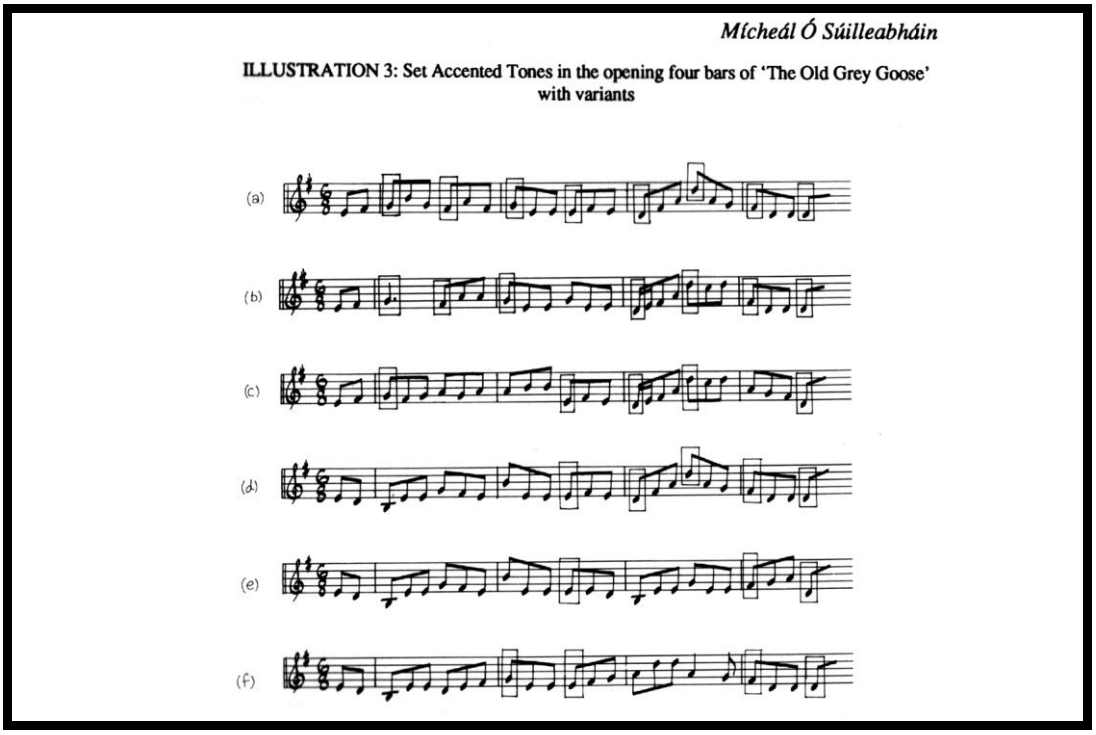


Fig.6.7 Illustration 3 from "The Creative Process in Irish Traditional Dance Music", p.123.

One could infer that each distinctive piece has a unique sequence drawn from the accented tones, which occur at the main accentual points in the piece. In other words this implies the existence of an equivalence classes of pieces with identical sequences of set accented tones. We do not know in advance which of the accented tones form the set accented tones. From Fig.6.7, it would appear that most of the accented notes participate, at least in the first part of the tune. We can seek a certain level of corroboration of the theory by seeing if all of the sequences of accented tones are unique, and hence "at the heart of the identity of the tune". If this is true, then if we extract the sequence of accented tones from each piece, and organise them in a frequency distribution, each such sequence should occur only once, unless the collection contains duplications. A simple program can be constructed to test this. The main part of this program is given below. The program and the following results are based on analysis of the accented tones only in the first parts of tunes. The program uses two classes to support the task. The instance of class **Store** denoted by 'store' is used to store an ordered collection of unique objects that are inserted into the store by the **put** member function. A frequency is associated with each stored object. The **PitchTuple** class stores tuples, and normalises the scalar quantities inserted so that the first component is taken as an origin, with a value of zero. Subsequent components are adjusted accordingly to the value of the tone relative to the number of

semitones it is distant from the first tone. Hence the following eight tones yield an 8-tuple as shown in Fig.6.8.

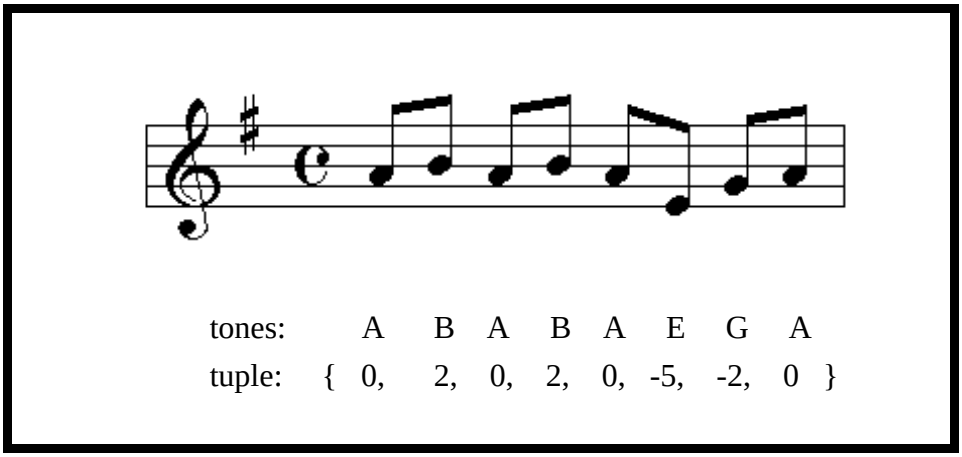


Fig.6.8 Pitch 8-tuple example.

Ex.4.1 Text.

.....these tones are at the heart of the piece's identity....

Ex.4.2 The Related Hypothesis.

Different tunes have unique sequences of accented tones.

Ex.4.3 Algorithm.

Visit each tune.
Calculate a tuple for the accented tones from the 'tune' part.
When all tunes are visited, examine each tuple for uniqueness.
Apply the decision criterion below.

Ex.4.4 Decision Criterion.

If two or more tunes share the same tuple, examine the tunes to see if they are closely related. Support the hypothesis if each different tune has a unique tuple.

Ex.4.5 Construction of Software.

The program uses two main classes, the **Store** class for holding tuples and the **PitchTuple** class that is used to represent and normalise the tuples. Grace notes are excluded from the calculations, for the same reasons as were given previously.[105]

[105] See section 6.8, ex3.3.

```
Store<PitchTuple> store;
store.init(100, tupleSize);
int countAll = 0;
String str, fname(argv[argc-1]);

while (getNextScoreNames(fname, str))
{
 Score s(str);
 ScoreIterator si(s, 0);
 countAll++;
 cout << "-";
 si.locate(BAR,1);
 PitchTuple tuple(tupleSize);
 int count = 0;

 while (si.getBarNo() != tupleSize/2 + 1 && ! si.isNullStave())
 {
  if ( si.getTag() == NOTE &&
   !(GRACE_NOTE < si.getAttributeSet()) &&
   ( si.getBarDist() == Rat(0,1) ||              // start of bar
     si.getBarDist() == (si.getTimeSig()/Rat(2))))// middle of bar
   tuple.put( si.getPitch12(), count++);
  si.step();
 }
 store.put(tuple);
}
```

Fig.6.9 Program for testing hypothesis of Ex.4.

Ex.4.6 Testing of Software.

Classes **PitchTuple** and **Store** have been tested individually. The complete algorithm is run on a random sample of 20 pieces from the corpus, and the results are checked manually.

Ex.4.7 Results.

```
Accented Tone analysis for pieces in =d:\mdb\crnh1\crnh1j.dir

Frequency    Pitch 16-Tuple
1            {0,12,5,0,-4,3,8,13,3,3,5,0,-4,3,8,10}
1            {0,9,10,12,0,9,10,7,0,9,10,12,17,19,10,5}
1            {0,7,5,10,0,7,5,5,0,7,5,10,14,12,5,5}
1            {0,7,4,7,0,7,2,4,0,7,4,7,14,7,2,4}
1            {0,7,2,5,0,7,10,2,0,7,2,5,7,7,10,2}
1            {0,7,0,5,0,7,10,5,0,7,0,5,3,2,10,2}
1            {0,5,9,7,12,16,12,10,0,5,9,7,12,17,10,5}
1            {0,5,9,5,2,5,9,7,2,5,9,5,2,5,9,7}
1            {0,5,8,0,0,0,3,0,3,0,5,8,0,0,0,-2}
1            {0,5,7,10,12,9,17,10,4,5,7,9,12,5,0,0}
1            {0,5,0,10,12,10,4,4,0,5,0,10,12,10,7,5}
1            {0,5,0,5,-2,-3,-2,-5,0,5,0,5,-2,2,-3,-7}
1            {0,4,4,4,0,5,-1,2,0,4,4,4,7,5,0,0}
1            {0,3,1,-5,0,3,1,-4,0,0,-2,-2,3,8,1,-4}
1            {0,3,0,3,-2,-7,-2,-7,0,3,0,5,9,3,0,5}
1            {0,2,0,9,0,2,-3,-2,0,2,0,9,0,0,-3,-7}
1            {0,2,-5,-8,-5,-8,-5,-12,0,2,-5,-8,-5,-12,-10,-12}
1            {0,1,3,1,-5,-9,-9,-7,0,1,3,0,8,1,-4,-4}
1            {0,0,15,12,7,12,7,3,0,0,15,12,7,12,7,5}
1            {0,0,12,14,9,7,2,4,-5,0,12,14,9,7,4,0}
1            {0,0,5,5,3,8,-2,-2,0,0,5,5,3,8,1,-4}
1            {0,0,4,2,9,12,9,7,0,0,4,2,9,12,2,0}
1            {0,0,2,4,-3,0,2,-3,-8,0,2,4,9,7,0,-3}
1            {0,0,2,2,0,0,10,2,0,0,2,2,5,12,9,2}
1            {0,0,2,-2,0,-2,7,7,0,0,2,-2,-2,-2,5,2}
1            {0,0,1,-2,0,0,1,-4,0,0,1,-2,12,10,1,-4}
1            {0,0,0,5,0,-5,-9,3,0,0,0,5,0,3,-5,-7}
1            {0,0,0,-2,0,0,-2,-2,0,0,0,-2,0,3,5,-2}
1            {0,0,0,-4,-4,-4,5,-2,0,0,0,-4,0,5,-4,-9}
1            {0,0,0,-7,-2,-2,-2,-9,-4,-2,0,8,5,0,0,-7}
1            {0,0,-2,-2,0,-2,-4,8,0,0,-2,-2,3,1,-2,-4}
1            {0,0,-2,-7,3,5,3,-3,0,0,-3,3,-7,-7,-7,-7}
1            {0,0,-5,-5,0,-3,-3,2,0,0,-5,-5,0,4,0,-3}
1            {0,-1,-3,4,0,-1,-5,2,0,-1,-3,4,0,7,-1,-1}
1            {0,-2,-4,8,3,0,0,-2,0,-2,-4,8,3,0,0,-4}
1            {0,-2,-4,5,3,10,12,3,0,-2,-4,5,3,3,0,-4}
1            {0,-2,-8,-12,-10,-2,-10,-12,0,-2,-3,-5,-8,-2,-8,-12}
1            {0,-2,-9,-5,0,1,3,7,0,-2,-9,-5,0,1,-2,-4}
1            {0,-3,0,3,-2,-5,3,3,0,-3,0,3,-2,3,-5,-7}
1            {0,-3,-3,-5,0,-1,4,7,0,-3,-3,-5,-1,4,0,-3}
1            {0,-3,-6,-6,0,-1,-3,7,0,-3,-6,-6,-3,6,4,-5}
1            {0,-4,1,-2,0,-4,1,-4,0,0,1,-2,0,1,-4,-4}
1            {0,-4,-9,-5,0,3,-5,-2,0,-4,-9,-5,0,3,-4,-4}
1            {0,-5,0,5,7,0,2,-3,0,-5,0,5,7,0,2,0}
1            {0,-5,0,-7,0,-5,-2,-9,0,-5,0,-7,-9,-2,-2,-7}
1            {0,-5,-3,-5,0,-5,-3,-6,0,-5,-3,-5,6,4,6,0}
1            {0,-5,-7,-7,0,-5,-7,2,0,-5,-7,-7,10,9,0,2}
1            {0,-5,-8,-5,-1,-1,-8,-3,-1,-5,-8,-5,-1,-3,-5,-5}
1            {0,-6,-9,-6,-12,-16,-12,-11,-9,-6,-9,-6,0,-6,-9,-4}
1            {0,-7,5,0,0,0,5,3,0,-7,5,0,-2,0,9,5}
1            {0,-7,0,5,-2,3,-2,-7,0,-7,0,5,-2,3,-7,-7}
1            {0,-7,0,0,0,-7,0,-5,0,-7,0,0,-2,3,-2,-5}
1            {0,-7,-2,-7,-12,-9,-4,-2,0,-7,-2,-7,-12,-9,0,-4}
1            {0,-7,-5,-7,0,-7,-5,4,0,-7,-5,-7,-7,5,0,-2}

Total number of pieces processed is 54
```

Table 6.8 Frequency distribution of tuples for CRNH1 using program of Ex.4.

Ex.4.8 Conclusions.

The results for the Breathnach collection of 56 jig pieces, in Table 6.8, show that they all have unique sequences of accented tones. This adds support to our hypothesis. The collection in question is modern and the author, by the use of a card index, ensured that no duplicate tunes were included. Such would have been unlikely in any case in such a small collection. Considerably more significant results may be obtained from running the analysis on the 365 pieces in the O'Neill collection, see table A3.1 in appendix 3. All of these entries proved unique except for two duplicated pairs. By manually scanning down through the sorted table of tuples produced by the analysis, we see immense diversity in the sequences of accentuated tones.

The next step is to examine the pieces corresponding to the pairs and to see if they are related. A small modification to the program that produced these results was made to give the user a facility for searching the corpus for instances of specific tuples. The modified version is given in appendix A2.3.

On running this program, the output produced shows that the two pieces that share the tuple { 0, 5, 0, 0, 0, 5, 9, 2, 0, 5, 0, 0, 5, 7, 9, 5) are

No 16 "ann do tinneas ne tae ta uait? - WHEN SICK IS IT TEA YOU WANT?
and
No 358 "imthigh do'n diabhal's corruidh tu fein - GO TO THE DEVIL AND SHAKE YOURSELF"

In spite of the different titles the music parts of these two pieces are identical. The lack of a thematic index led O'Neill to include the same music twice.

The tuple { 0, 3, 5, 3, 0, 3, -2, -2, 0, 3, 5, 3, 0, 1, -4, -4 } appears twice, first in

No. 42 "Biodhg suas liom - MOVE UP TO ME"
and also
No. 325 "bo leath-adharcach uí mhartain - MARTIN'S ONEHORNED COW"

In this case, the music has a number of differences, No. 42 is pitched a perfect fourth higher, it has a different key signature and has some minor differences in the unstressed notes. They are clearly very closely related.

The sequence of accented tones is, in effect, a normalised pitch vector, which is tied closely to the identity of the tune in the sense that different tunes have unique vectors.[106] A further study is needed to check the validity of this from another perspective. We need to establish whether closely related tunes have pitch vectors that are similar or possibly identical.

To summarise, we have shown here that -

In general there is an immense diversity of sequences of accented tones.

The only case of sharing of sequences of accented tones between tunes is found for closely related tunes.

We have not shown however, that all related tunes have the same or similar sequences of accented tones.

[106] A similar technique, that of extracting stressed pitches, is used by Helmut Schaffrath, op.cit., for information retrieval purposes.

Chapter 7. Applications - Investigatory Analyses.

The last chapter was concerned with verifying statements that a musicologist made about a corpus of music. In this chapter, examples are presented showing the potential of ***scoreView*** *for carrying out investigations on a corpus.*

The main difference between the kind of inquiry that uses a computer and one that is done manually, arises from the ability of a computer to act as a speedy and tireless amanuensis which excels in some tasks, in particular in tasks of a combinatorially intensive nature. This pushes out the limits of what it is feasible to do, given that humans have limitations to their energy, attentiveness, accuracy and time. It may be recalled that one of the conditions for a set of instructions to be an algorithm is that it is capable of being done with a pencil and paper (see 4.2). The computer scores over the pencil and paper in situations where the work would take too long and/or be too tedious and/or where manual results might be too unreliable.

Five examples of the use of the system are given below under the following headings.

Scale Finding: The first example shows how we can find the types and frequencies of scales that are present in the corpus.

Feature Extraction: The second example illustrates how we might extract and organise information about a melodic feature of double jig tunes.

Melodic Difference: The third example illustrates how we might construct algorithms to calculate various numerical estimates of melodic difference between two segments of music. A number of developments of the basic algorithm are discussed and some of these are implemented.

Form and Exhaustive Search: The fourth and fifth examples illustrate ways in which a melodic difference algorithm might be used to extract meaningful information from the corpus. The fourth example is concerned with an evaluation of 'crude' melodic forms of corpus members, and the fifth example

concerns itself with exhaustive searches of the corpus for identifying exact copies and close variants.

This thesis does not undertake a comprehensive analysis of the corpus of Irish double jig tunes. The algorithms presented here are primarily intended to illustrate how **scoreView** might contribute towards such an analysis.

7.1 Scale of a Double Jig.

This section demonstrates a method by which the scale of a piece of music may be identified and uniquely labelled. Here the word 'scale' is being used in a very restricted sense.[107] What is meant here is simply the fundamental intervallic pattern of the set of note classes used in the piece. This section is not about finding which note of the scale is the modal one, but instead, it is concerned with working out and classifying the basic intervallic relations in the scale of a piece. We can view this process as a procedure which traverses all the notes in a piece and forms a set of all the pitch classes encountered. In order to identify the scale, these sets of pitch classes have to be mapped into a standard form which preserves the intervallic relationships of the scale. The requirements for this standard mapping are that

- all versions of the same scale in any key should map to the same standard form,

- no two different scales should map to the same standard form.

The pitch class set is an appropriate tool for cumulating and recording the set of pitches in a piece. The pitch class set, as proposed by Forte[108] is nothing more than the mathematical notion of a set of elements, where the elements are chromatic pitch numbers, or their modulo 12 equivalents. In order to compare two sets, and to identify if they are made up of the same intervallic material, Forte proposes a number of transformations which reduces any possible set of pitch classes to 220 distinct sets called prime forms. He also provides standard labels for them. The basis on which

---

[107] Stanley Sadie The New Grove Dictionary of Music and Musicians volume 16 (London: MacMillan 1980) has the following definition of a scale. by William Drabkin: "A scale is a sequence of notes in ascending or descending order of Pitch." The usage here corresponds with this definition, and not as further refined in Grove - "As a musicological concept, a scale is long enough to define, unambiguously a mode, tonality, or some special linear construction, and that begins and ends (where appropriate) in the fundamental note of the tonality or mode;...".

[108] Allen Forte The Structure of Atonal Music (New Haven and London: Yale University Press 1973).

this mapping takes place, and the individual mappings involved are given below. It will be demonstrated, that Forte's prime forms are inappropriate for the task of scale classifications, but that a closely related mapping accomplishes the task satisfactorily.

Forte lays out, in steps 1 to 3 below, the axioms under which pitch class sets may be transformed while still retaining their basic identity. Steps 4, and 5 below reduce the sets to a standard form, using the preceding axioms. This standard form may appear on Forte's list of prime forms. If it does not, one further series of transformations are introduced, the first of which is based on an axiom of inversional equivalence. This is followed by a repeat of transformations 4 and 5. These steps are described in step 6 below.

1. The axiom of **octave equivalence**, states that change of register does not affect notation-class membership. Hence pitch 0 is equivalent to 12, -12, 24 and -24, for example. Also pitch 1 is equivalent to 13, 25, -11 and -23.

In the case of the scales under study, each scale repeats its intervallic pattern in upper and lower octaves, and hence, for the purpose of scale identification, transformations that use octave equivalence maintain the basic scale structure.

2. Enharmonic notes are equivalent. Hence C sharp is equivalent to D flat, and either can be represented by the same pitch class element, which is also equivalent to the number 1.

In the corpus under study, the music is modal, notes outside of the diatonic scale are rare, and where they do occur, issues of enharmonic equivalence do not arise in any practically important way.

3. Normal ordering is achieved by successively rearranging the ordered set in ascending order. All circular permutations of the set, with the addition of 12 to a shifted element, are regarded as being equivalent normal orderings. For example the normal order of the set { 2 0 5 7 11 9 4 } is { 0 2 4 5 7 9 11 }. This is also equivalent to { 2 4 6 7 9 11 13}.

Note sequence has no effect on determination of scale. The addition of 12 to a set element is covered under 1 above.

4. The main step for mapping a set into its prime form is done in Forte's classification system using the following algorithm, which employs only transforms of 1-3 above.

Select the ordered set with the least difference between the first integer and the last from the various circular permutations. In the case of a tie, select the permutation with the least difference between the first and second element. If this is the same for more than one permutation, select the permutation with the least difference between the first and third element, and so on, until the difference between the first and the next to last element has been checked. If all these differences are the same each time, select one ordering arbitrarily as the normal order.

Permuting the notes of a scale in this way, simply changes the order, but leaves all relevant intervallic relativities of the set unchanged.

5. By the transposition operator, which adds a positive or negative constant to every element in a set, one can, in effect, produce a class of transpositionally equivalent sets. The set from step 4 is made into a standard representative by the application of the transposition operator so as to make its first element zero.

Transposing the notes of a scale in this way, changes only the key of the scale, and leaves all the intervallic relativities, and hence the type of the scale, unchanged.

6. Forte goes one step further, a step that may be required is some cases to arrive at his prime forms. He proposes the inverse transformation by which pitch numbers are transformed into their inversional equivalents. These are represented below -

0 <-> 0
1 <-> 11
2 <-> 10
3 <-> 9
4 <-> 8
5 <-> 7
6 <-> 6

When a set is transformed according to the above mapping, the set is subsequently normalised as in 1 - 5 above.

If we want to classify a set of pitches according to Forte's scheme, we first perform transformations 4 and 5 on the set of pitches, and then look up the resultant set in a table of prime forms to find a match. If the match is absent from the table, we perform transformation 6, followed by steps 4 and 5 above on the transformed set, and we are then guaranteed that our calculated form will be present in the table of prime forms. We can then find its associated name.

It tuns out that, if we omit one of Forte's transformations, the invertional transposition of no.6 above, we will arrive at a unique characterisation of the underlying scale of pieces. The cost of doing this is that we increase the number of possible prime forms by a factor less than two.

If transformations 4 and 5 are applied to all the notes in a scale, we find that we do not alter the basic intervallic relations in such a way as to change the identity of the scale. If however we apply transformation 6 as well, this will cause unlike scales to map to the same prime form. An example of this occurrence follows, together with a definition of a new special prime form, called a **non-inversionally equivalent** (NIE) prime form.

Non-Inversionally Equivalent (NIE) Prime Form is defined here as the set of standard pitch classes together with their names, that any pitch class set is transformed to under transformation 4 and 5 above. In identifying and naming these sets we simply use the Forte prime forms and the associated name wherever we can. If a set, when transformed under step 4 and 5 above, fails to appear in Forte's table, then its transformation under 6 followed by 4 and 5 must, we use the same prime form name as Forte, but distinguish it by prefixing it with the letter I followed by a hyphen. Hence corresponding to Forte's name 3-7, we have two non-invertionally equivalent prime forms with names

3-7 for set { 0 3 7 }
and I-3-7 for set { 0 4 7 }

The processes involved here can be illustrated by two 3-note scales.

The prime form of the scale D F A is got in the following stages

1. Express as a pitch-class set { 2 5 9 }

2. Consider the various rotations { 2 5 9 } { 5 9 14 } { 9 14 17 } and select the one with minimal distance between the first and last, that is 9 -2 = 7, 14 -5 = 9, 17 - 9 = 8. Hence we select the first one.
3. Transpose, to make the first element 0, gives { 0 3 7 }.
4. Consult Forte's list of prime forms and we find its name is 3-7.
   We also use the name 3-7 as the NIE prime form.

If we now repeat this for another distinct 3-note scale, that of C E G, we get

1. Express as a pitch-class set : { 0 4 7 }
2. Consider the various rotations { 0 4 7 } { 4 7 12 } (7 12 16 } and select the one with minimal distance between the first and last element, that is 7 -0 = 7, 12 - 4 = 8, 16 - 7 = 9. Hence we select the first one, as it has the smallest difference.
3. Transpose to make the first element 0 gives { 0 4 7 }
4. Consult Forte's list of prime form, we find it is not present.
5. When we invert the set { 0 4 7 } we get the set { 0 8 5 }
6. Put into normal form { 0 5 8 }
7. Consider the various rotations { 0 5 8 } { 5 8 12 } { 8 12 17) and select the one with minimal distance between the first and last element, that is 8 -0 = 8, 12 - 5 = 7, 17 - 8 = 9. Hence we select the second one.
8. Transpose, to make the first element 0 gives { 0 3 7 }
9. Consult Forte's list of prime forms and we find its name is 3-7. The NIE prime form name is I-3-7.

Here we have taken two distinct scales and mapped them into the same prime form of Forte! This illustrates that the Forte classification scheme will not do, but if we remove the inversion mapping, and distinguish the two resulting separate prime form names, we will uniquely characterise the scales.

In terms of the NIE prime form we find that

Scale D F A maps to NIE prime form 3-7 for set { 0 3 7 }
Scale C E G maps to NIE prime form I-3-7 for set { 0 4 7 }

It would be instructive, at this stage to speculate on which NIE prime forms we might expect from an examination of the pitches in a double jig tune. The heptatonic major scale { 0 2 4 5 7 9 11 } which maps to 7-35 without using the inversional transposition, is one such candidate scale. A list of some scales that might be expected to crop up are given below, in the key D major

D E #F G A B #C	heptatonic	7-35
D E #F G A B	hexatonic	6-32
D E #F A B #C	hexatonic	6-32
D #D E #F G A B #C	8-note[109]	8-22
D E #F G A B C #C	8-note	8-23
D E #F G A	pentatonic	5-35

Note that the two hexatonic cases above are in effect the same scale. This is because they are transpositionally equivalent. Here we are not considering modality. The first 8-note scale above arises mostly in piping, as the chanter has an extra #D. The next 8-note scale illustrates the presence of both C and #C in the same tune.

```
String origFilename(argv[argc-1]);
String currentFilename;
Store<PitchClasses> spcs(20);

while ( getNextScoreNames(origFilename, currentFilename))
{
 Score s(currentFilename);

 ScoreIterator si1(s);
 ScoreIterator si2(s);

 si1.locate(BAR,1);
 si2.locate(BAR, SCANLENGTH); // start of 8th bar

 do si2.step();
  while ( si2.getBarDist() < (si2.getTimeSig()*Rat(1, 2)) &&
       si2.getBarNo() == SCANLENGTH );

 PitchClasses pcs;
 pcs.pitchClass(si1, si2);
 pcs.NIEPrimeForm();
 spcs.put(pcs);
}
```

Fig.7.1 Scale classification program.

[109] In the current context, the term 'octatonic' is avoided because of its associations with a specific 8-note scale.

The output from running this program on the 54 double jig tunes in Breathnach's Ceol Rince na hEireann is given in Table 7.1.

Distribution of scales for file(s) =d:\mdb\crnh1\djig.dir

Prime Form	Name	Frequency
{0 1 2 3 5 7 8 10 }	8-23	3 (6%)
{0 1 3 5 6 8 10 }	7-35	18 (33%)
{0 1 2 4 6 7 9 }	7-29	1 (2%)
{0 2 4 6 7 9 }	I-6-33	5 (9%)
{0 2 4 5 7 9 }	6-32	18 (33%)
{0 2 3 5 7 9 }	6-33	1 (2%)
{0 2 4 7 9 }	5-35	5 (9%)
{0 1 3 5 6 8 }	6-Z25	2 (4%)
{0 2 4 5 7 }	I-5-23	1 (2%)

Table 7.1 Distribution of scales for CRNH1.

Observations.

It can be seen here that most of our predictions are confirmed. Significant norms here include 7-35 and 6-32 which between them, account for the majority of tunes (66%) , with a small number for the most common pentatonic scale, 5-35 (9%). The predicted 8-note scale 8-22 did not occur.

Tune no. 44 has key signature of G major. The note B is absent, and hence it is basically a hexatonic scale, but it carries both variants of C (C and #C), yielding a NIE prime form of 7-29.

Prime form I-6-33 corresponds to tunes nos. 23, 29, 34, 38 and 45. No. 23 has key signature of G major, but with the note #F absent, but with #C appearing in it instead of C. Nos. 29, 34, 38 and 44 have key signature of G major, with note B omitted. Interestingly, in no. 34 the note B does occur in its initial anacrusis, but it is absent from the tune proper and from the turn. I-6-33 corresponds to the normal diatonic major scale with the third of the scale omitted.

A comparison with the output from O'Neill's, in table 7.2, shows very interesting variations in the occurrences of scales. The greater diversity of cases with low frequencies is due, to a large extent, to misplaced accidentals in tunes which are not written out correctly, seemingly because of the difficulty that O'Neill's transcriber had in dealing with key signatures other than D and G major. A detailed study of the O'Neill sources would be required, in order to iron out many of these problems. Note that the percentage frequencies have been rounded to the nearest whole number,

resulting in showing occurrences of 1 out of 365 as having a frequency of 0%. One interesting fact emerges about the percentage of heptatonic tunes based on the diatonic major scale is that it differs significantly between the two collections. This can be seen from NIE prime form 7-35 which occurs in 33% of tunes in Breathnach's compared with a 58% occurrence in O'Neill's. Clearly further investigation is called for here.

```
Distribution of scales for file(s) =d:\mdb\tdmoi\djig.dir

Prime Form                    Name              Frequency
{0 1 2 4 5 7 9 10 }           8-26              2     (1%)
{0 1 2 3 5 6 7 8 10 }         9-9               1     (0%)
{0 1 2 3 4 5 7 8 10 }         9-7               2     (1%)
{0 1 2 3 5 7 8 10 }           8-23              14    (4%)
{0 1 2 3 5 6 8 10 }           8-22              3     (1%)
{0 1 3 5 6 8 10 }             7-35              213   (58%)
{0 2 4 5 7 8 9 }              I-7-27            2     (1%)
{0 1 2 4 5 6 7 9 }            8-14              1     (0%)
{0 2 4 5 6 7 9 }              I-7-23            1     (0%)
{0 1 2 4 6 7 9 }              7-29              1     (0%)
{0 2 4 6 7 9 }                I-6-33            6     (2%)
{0 1 2 4 5 7 9 }              7-27              1     (0%)
{0 2 4 5 7 9 }                6-32              84    (23%)
{0 2 3 5 7 9 }                6-33              1     (0%)
{0 1 2 4 7 9 }                6-Z47             1     (0%)
{0 2 4 7 9 }                  5-35              8     (2%)
{0 1 3 5 7 8 }                6-Z26             1     (0%)
{0 1 3 5 6 8 }                6-Z25             18    (5%)
{0 1 3 5 8 }                  5-27              2     (1%)
{0 2 4 5 7 }                  I-5-23            3     (1%)
```

Table 7.2 Distribution of scales for TDMOI.

Prime Form Notation Extensions.

In order to allow for a uniform printing of pitch class sets in later examples, the following names are used for sets of one or two notes, which Forte does not list.

Set	Name
{ 0 }	1
{ 0, 1 }	2-1
{ 0, 2 }	2-2
{ 0, 3 }	2-3
{ 0, 4 }	2-4
{ 0, 5 }	2-5
{ 0, 6 }	2-6

Table 7.3 Extensions to list of prime form names.

7.2 The Initial Anacrusis in Double Jigs.

In the next analytic example the process of extracting and analysing the structure of small melodic features is demonstrated. Such melodic features that we might want to study include the melodic structure at cadence points, or the occurrence of certain common melodic formulas. In this example the subject of attention will be the initial notes of tunes.

In many jigs, the initial stressed note may be preceded by one or more notes that serve a number of functions. As these are the first notes of the tune, they provide initial cognitive clues to the listener about the type of music. Apart from possible practical uses for musician-dancer interaction, they have a number of significances of a musical nature. In relation to pitch, they narrow down the possibilities of the scale of the piece. Also, taken in conjunction with the first accented note of the tune, they give the early clue about its mode. They set the initial pace of the tune, by establishing possibilities for a tactus. Also, in relation to the overall phrasing of a tune, they play an important part. Tunes which start with an anacrusis, normally perpetuate a phrasing of exactly the 8-bars in length through all of subsequent parts, with the turn of the piece having a similar anacrusis, and likewise at the start of any additional part. Hence if we wish to study the development of a listener's sense of modal centre, or of a listener's sense of tactus, or of a listener's sense of phrasing, a study of the anacrusis part of the first 8-bars, the tune part, of a tune is important. At the outset a number of questions may be posed. What sequences of pitches are allowable in a valid anacrusis? Which are the most common sequences? What time values are associated with each of these notes? What scales might be implied by these?

Answers to some of the above questions can be got by means of an algorithm. The use of the PitchTuple class and the Store class greatly simplifies this task. The code is given in Fig.7.2.

```
String str;
int countInitials[MAXNOTES] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

// pcst is an array of class Store<PitchClasses> for storing
// prime form sets of initial components. Each array element here
// stores prime forms of different cardinalities
Store<PitchClasses> pcst[MAXNOTES];

// tst is an array of class Store<PitchTuple> for storing
// tumple of initial components.
// Each array element here stores tuples of different sizes
Store<PitchTuple> tst[MAXNOTES];

// Specilize the elements to different pitch class and tuple sizes
for ( int count = 0; count <MAXNOTES; count++)
{
 pcst[count].init(100, count+1);
 tst[count].init(100, count+1);
}
int countScores = 0;

while (getNextScoreNames(origFilename, str))
{
 Score s(str);
 if ( ! s.isNull() )
 {
  ScoreIterator si(s);
  countScores++;
  int count = 0;
  PitchClasses pcs;
  int countNotes = -1;
  int pitchStore[MAXNOTES];

  while (si.getBarNo() == 0)
  {
   if ( si.getTag()== NOTE)
   {
    pcs.pitchClassInc(si);
    count++;
    pitchStore[++countNotes] = si.getPitch12();
   }
   si.step();
  }

  // next step to next note and add to our stores
  while ( si.getTag() != NOTE) si.step();
  pcs.pitchClassInc(si);
  pitchStore[++countNotes] = si.getPitch12();

  PitchTuple tpl(countNotes+1);
  for ( int cnt = 0; cnt <=countNotes; cnt++)
   tpl.put(pitchStore[cnt], cnt);
  tst[countNotes].put(tpl);
  countInitials[count]++;
  pcs.NIEPrimeForm();
  pcst[count].put(pcs);
 }
}
```

Fig.7.2 Program to extract initial anacrusis details.

```
Analysis of Initial Notes
File: =d:\mdb\crnh1\djig.dir
54 scores processed

NIEPrimes/Tuples          Frequency
=================================================================
Nr. of Notes:1            14   25%
_________________________________________________________________
NIEPF:{0 } 1              14   25%
_________________________________________________________________
Tuple:{0}                 14   25%
=================================================================
Nr. of Notes:2            18   33%
_________________________________________________________________
NIEPF:{0 5 } 2-5          7    12%
NIEPF:{0 4 } 2-4          1    1%
NIEPF:{0 3 } 2-3          1    1%
NIEPF:{0 2 } 2-2          7    12%
NIEPF:{0 } 1              2    3%
_________________________________________________________________
Tuple:{0,5}               3    5%
Tuple:{0,2}               3    5%
Tuple:{0,0}               2    3%
Tuple:{0,-2}              4    7%
Tuple:{0,-3}              1    1%
Tuple:{0,-4}              1    1%
Tuple:{0,-5}              1    1%
Tuple:{0,-7}              3    5%
=================================================================
Nr. of Notes:3            21   38%
_________________________________________________________________
NIEPF:{0 4 5 } I-3-4      1    1%
NIEPF:{0 2 5 } 3-7        4    7%
NIEPF:{0 2 4 } 3-6        6    11%
NIEPF:{0 2 3 } I-3-2      2    3%
NIEPF:{0 1 3 } 3-2        7    12%
NIEPF:{0 3 } 2-3          1    1%
_________________________________________________________________
Tuple:{0,4,5}             1    1%
Tuple:{0,2,4}             3    5%
Tuple:{0,2,3}             1    1%
Tuple:{0,-1,-3}           1    1%
Tuple:{0,-2,-3}           7    12%
Tuple:{0,-2,-4}           3    5%
Tuple:{0,-3,0}            1    1%
Tuple:{0,-3,-5}           4    7%
=================================================================
Nr. of Notes:4            1    1%
_________________________________________________________________
NIEPF:{0 2 5 7 } 4-23     1    1%
_________________________________________________________________
Tuple:{0,2,5,7}           1    1%
=================================================================
```

Table 7.4 Initial anacrusis details for CRNH1.

In the panels in the above, which are delimited by a line of '='s, an analysis is given for each case of one note, that is of the first stressed note only or, in other words the case of no anacrusis; two notes are given for the case of one note before the first stressed note; three notes are given for the case of two notes before the first stressed

note, etc. Each panel is headed by its absolute and relative frequency of occurrence. Here we see that 25% of tunes in CRNH1 have no anacrusis. 33% and 38% of tunes have respectively, one and two notes before the first stressed note. No case is found in this collection with more than three notes before the first stressed note, and only a single case is found with exactly three notes preceding the initial stress. If we look at the panel which shows the two note case, that is one note before the first stressed note, we see that tuple { 0, -2 }, representing a falling major second is the most commonly occurring interval. There are only eight ways in this collection of providing an initial single note lead in. There are more cases of falling first intervals than rising ones, and surprisingly, a rising minor second is not present in any tune. Also, in the case of a single note anacrusis, the only falling intervals are perfect fifths, perfect fourths, major and minor thirds and major seconds. When run on the much larger O'Neill collection, we get the results shown in table A3.2 in appendix 3.

In both collections approximately 50% of tunes have either no anacrusis or a single note one. There is a significant difference in the percentage of tunes that have no anacrusis between CRNH1(25%) and TDMOI(16%). Most of the patterns that appear in CRNH1 also appear in TDMOI.

It is likely that an interesting relationship can be found between the structure of the anacrusis and the scale of a piece. Note that if the scale in question is gapped, then the concept of a ‘consecutive note’ involves intervallic possibilities of minor and major thirds as well as minor and major seconds and hence further analysis is required to identify patterns that move consecutively. The inclusion of non inversionally equivalent prime forms in the printout facilitates further study of the initial tonal relationships.

7.3 Crude Melodic Similarity or Difference Algorithms.

One of the important areas that computers have been used is in extensively searching for instances of melodic borrowings.[110] In folk music, two written

[110] In a recent article by E. Selfridge-Field: "Music Analysis by Computer" in Goffredo Haus Music Processing (Oxford 1993), p.3, a review of the activities in music analysis is classified into six categories, she says

> " The activities to be discussed fall into five areas of concentration - (1) linguistic analogy; (2) attribute description using statistical methods, (3) repertory-specific studies; (4) theory-specific implementations; and (5) style-specific simulations. A sixth and very important area of activity - similarity studies - intersects the other but must be excluded here because its inclusion requires more detailed consideration than space permits and because in studies to which it is central the data on which it depends often consists of very small samples from very large numbers of works. This sets it quite apart from studies that deal in a more comprehensive way with smaller groups or single features of works."

transcriptions of a tune are unlikely to be identical. Hence a direct comparison of the notes of two tunes transcribed from two performances of the same piece will inevitably reveal some differences. Algorithms which deal with the problem of identifying exact and near versions of two melodic segments are spoken of as melodic difference or melodic similarity algorithms. The first of the following examples illustrates a similarity measure based on contour information. This measure brings identical tunes together, irrespective of the key in which they were notated, but it has a number of shortcomings. All algorithms that deal with interval information only, share an inability in handling similar rather than identical tunes. Additionally they ignore perceptually significant information of a durational and metrical kind. Most of the following sections present an alternate approach which was first proposed in 1972.[111] All of the difference measures discussed assume that the issue of segmentation has already been tackled. In the case of dance music, the bar is taken as the smallest unit of segmentation. This is an approximate, but reasonably effective solution to the segmentation problem for the current corpus. All of the difference algorithms totally ignore any structure within the tune segment under comparison. They rely on general principles of music perception, but are not deeply based on cognitive theory. For this reason they are referred to as 'crude' difference algorithms. They are of use as an initial mechanism of searching for melodic variants. This approach differs significantly from the more sophisticated one of Mongeau and Sankoff[112] who develop a difference measure in terms of consolidation and fragmentation. They claim that their method will detect melodic differences in line despite gross differences in key, mode and tempo. The methods presented here have the advantage over Mongeau's and Sankoff's in being computationally more efficient as they do not involve combinatorially extensive processing.

Some of the early history of studies are given in op.cit. Stephen Dowland Page, pp.35 - 35.

> "One of the earliest approaches was to produce interval vectors or sequences for each tune and to arrange these vectors in an ordered sequence. The use of interval vectors get over the problems associated with trying to compare two tunes in different keys. The ordered sequence succeeds in bringing together identical tunes. However it falls down badly in bringing together variants, except in cases where the variants occur at the end of each vector. This scheme was use by Benjamin Suchoff "Serbo-Croatian Folk Songs", see op.cit. Harry Lincoln, pp. 193-206. Some of the basic limitations inherent in the system were overcome in the Suchoff study by the development of a program to extract sub sequences from each sequence of intervals. One additional problem associated with the representation of a tune as a sequence of intervals arises from the absence of temporal and metric information."

[111] Donncha O Maidin "Computer Analysis of Irish and Scottish Jigs" Baroni and Caglione, op.cit., 1984, pp.329-336

[112] Marcel Mongeau and David Sankoff "Comparison of Musical Sequences" in Computers and the Humanities volume 24(1990), pp.161-175.

All of the comparison algorithms given below treat the segments outside of their original contexts. They evaluate melodic difference numerically. This number can be thought of as a measure of the distance between two melodic segments. Such distance measures, when used for comparing two segments of duration **r**, have a number of properties. Theses include

difference(si1, si2, r) >= 0,

difference(si1, si1, r) == 0,

difference(si1, si2) == difference(si2, si1).

where **difference** is a function which returns the numerical distance between two melodic segments of duration **r**, starting at positions **si1** and **si2**, respectively.

7.3.1 Intervallic Based Difference Measures.

Intervallic comparisons were used in the earliest difference studies in computational musicology. Benjamin Suchoff[113] used an interval sequence approach to compare segments from Bartok's Serbo-Croatian folk songs. Whereas reducing tunes to intervals overcomes the problem of bringing together identical segments from different keys, a serious problem arises in identifying variants. Suchoff's solution was to compare substrings of the interval sequence for each tune. One problem of this approach lies in the potentially combinatorially explosive possibilities for forming different strings for comparison, especially where the melodic segments under comparison are long. Richard E. Overill[114] deals with this computational complexity in the comparison of interval sequences by applying techniques of Approximate String Matching (ASM) to the problem.

Interval comparison is a trivial program to implement in **scoreView**. The following section illustrates a variation which takes a very simple approach to the comparison of melodic segments based on contour information.

7.3.2 Melodic Difference Algorithm with Contour Information.

---

[113] Benjamin Suchoff "Computer Oriented Comparative Musicology" in Harry Lincoln, op.cit. 1970, pp.192-205.

[114] Richard E. Overill "On the Combinatorial Complexity of Fuzzy Pattern Matching in Music Analysis" Computers and the Humanities volume 27 (1993), pp.105-110.

Contour information can be viewed as a reduction of intervallic information to three states. These are rising, falling and stationary. These contour states may in turn be combined into 5 possible different states which represent the juxtopositioning of contour information between two melodic segments. These states are

w_1 = similar motion, i.e. both rising or both falling
w_2 = contrary motion, i.e. one rising and one falling
w_3 = one stationary, one moving
w_4= both stationary
w_5= undefined

If we assign weights to these states, we could select the weight 2 for contrary motion between corresponding contours(w2), 1 as a weight if one melody is stationary and the other moving(w3), and zero for all other weights(w1, w4 and w5).

Next we must define which notes from each segment participate in these comparisons. We do this by placing the melodic segments together in time sequence. The time axis is divided into time-windows where each window represents the longest time for which both melodic segments have a uniform activity. This is illustrated over in Fig.7.3(c). The sections of the green line represent window durations which are used as weights.

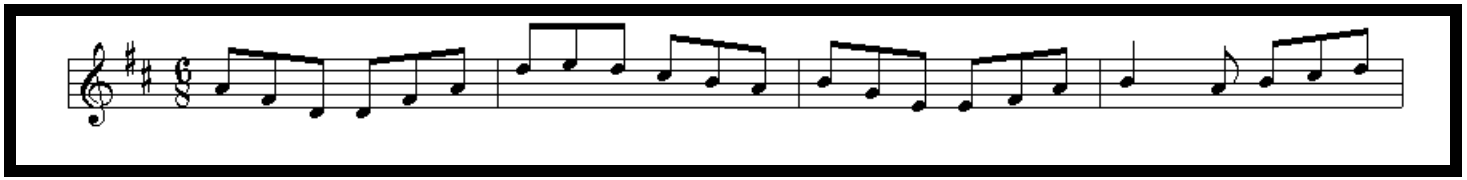


Fig.7.3(a) Start of 'Shandon Bells' from TDMOI.

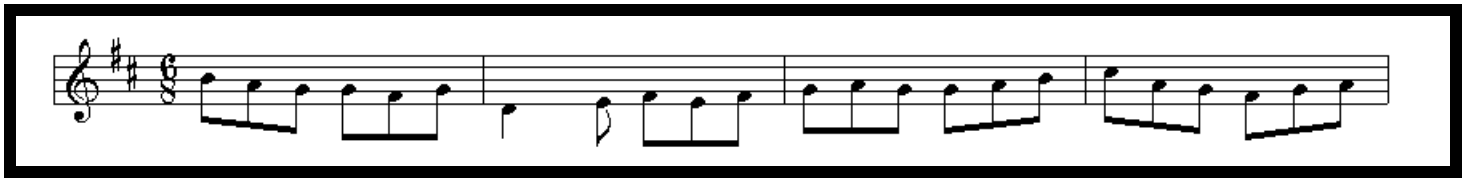


Fig.7.3(b) Start of 'The Yellow Flail' from TDMOI.

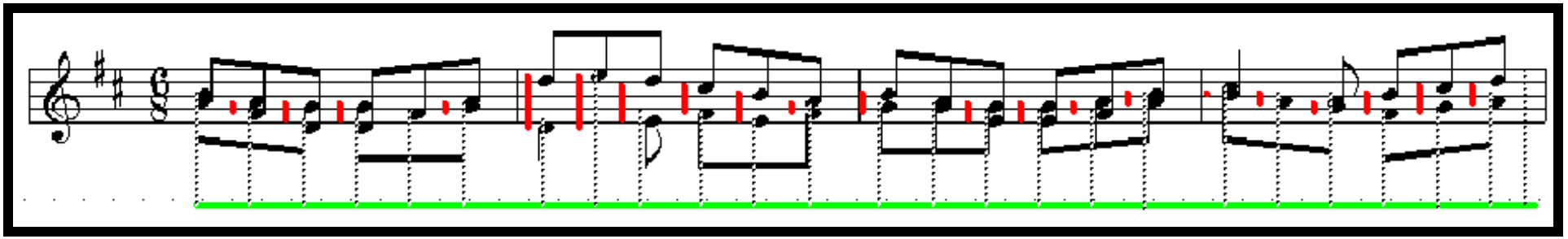


Fig.7.3(c) Segments of 7.3(a) and 7.3(b) are juxtaposed in time window order.
The dotted vertical lines segment the green line into divisions, each one of which represents a window.

This approach will be used in all of the comparison algorithms in this section. It works well for the music under study, but it may need modifications in analysis of genres where related melodic segments have certain kinds of rhythmical or durational variation.

Some of the one-bar segments shown in Fig.7.4 are used for illustrating the operation of various difference algorithms.

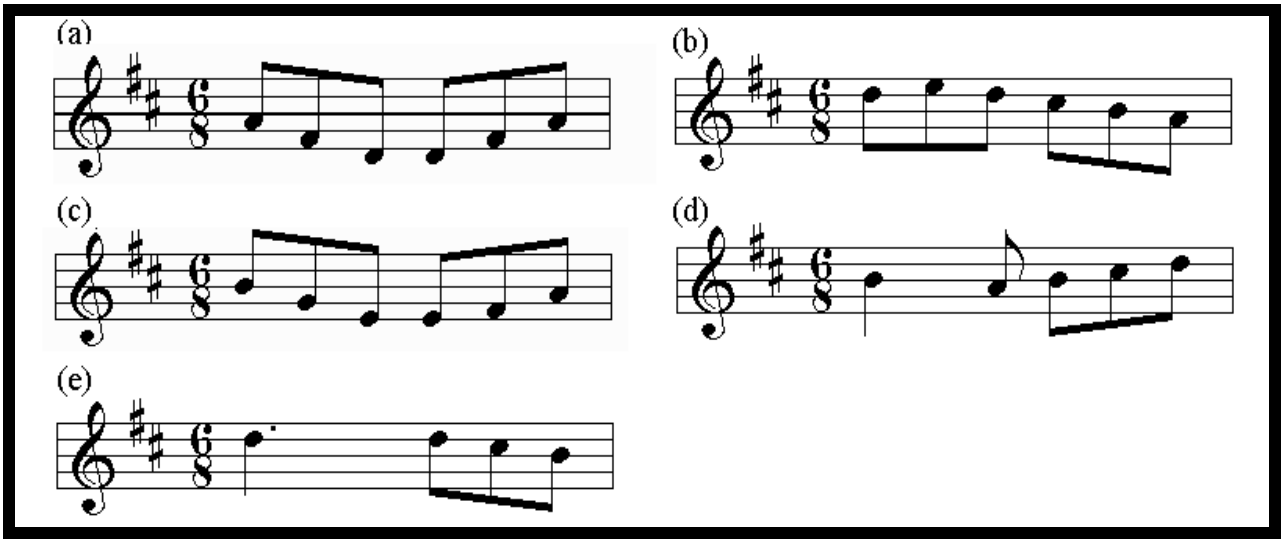


Fig.7.4 Sample melodic segments for illustrating difference algorithms.

The result of running the contour based difference algorithm is shown in Table 7.5.

	a	b	c	d
b	**7**			
c	**0**	**7**		
d	**2**	**7**	**2**	
e	**6**	**3**	**6**	**6**

Table 7.5 Differences calculated from contour information only.

This simple algorithm works well in some cases above. It detects the relationship between **a** and **c**, but gives rather unsatisfactory results in comparing **a** with **d** and **c** with **d**, both of which yield the second smallest score of 2.

The following algorithms use pitch instead of interval of contour information. One apparent disadvantage is that these algorithms fail in identifying transpositionally equivalent segments. Another approach to the transposition problem is to convert the

representation of pitches to a common tonic. Is such schemes[115], identifying the tonic will have to be done manually. This approach is likely to run into difficulties where the tonic is not uniquely identifiable, and in cases where it is desirable to make comparisons between segments at different diatonic transpositions. We will see, in 7.3.6, there is a technique for overcoming this.

7.3.3 Simple Window Weighted Melodic Difference Algorithm.

An algorithm that measures the difference between note pitches can be viewed roughly as calculating the sum of the lengths of the red lines in the Fig 7.3. This algorithm has is origin in the idea of representing musical pitch geometrically.[116]

Intuitively is seems wrong that difference measures should be influenced equally by comparisons between pairs of long notes as it is for comparisons between pairs of short notes. However, if the length of each individual pitch difference is weighted according to the width of the window to which it belongs, we make a provision for this inequality. This ensures that a melodic segment that carries a series of short notes, will not contribute unduly to the difference estimate.

The calculations involved are expressed, as follows -

Suppose we have **n** windows, which we label with integers from **1** to **n**.
Let $\mathbf{p_{ik}}$ be the pitch expressed in chromatic pitch numbers for window **k** of tune **i**. Then,

$$\textbf{Difference} = \sum_{k=1}^{n} |p_{1k} - p_{2k}| \, w_k$$

where $\mathbf{w_k}$ is the weight attaching to this difference and in this case the weights are equal to the widths of the corresponding windows.

The running of this algorithm on various combinations of the following melodic segments yields[117] results shown in Table 7.6.

---

[115] Martin Dillon and Michael Hunter "Automated Identification of Melodic Variants in Folk Music" Computers and the Humanities, volume 16 (1982), pp.107-117.

[116] Carol L. Krumhansl Cognitive Foundations of Musical Pitch (Oxford 1990), pp.112-119.

[117] The full matrix is symmetric one about a diagonal of zeroes.

	a	b	c	d
b	**538**			
c	**88**	**450**		
d	**438**	**275**	**350**	
e	**575**	**88**	**487**	**213**

Table 7.6 Window-weighted melodic difference results.

This algorithm, in common with all simple algorithms, gives a difference measure of 0 if both melodic segments are identical in pitches and in durations. When such an algorithm is used for identifying melodic segments that are similar, rather than identical we may have some reservations. It fails to take account of important cognitive factors such as metrical stress, which, as we saw was so important in the case of set accented tones in the last chapter. We see that segments **a** and **c** are close with a difference of 88. However a less satisfactory aspect is that segments **b** and **e** are evaluated as being equally close.

The next section shows how a further improvement can be made by incorporating information for metrical stress.

7.3.4 Melodic Difference Algorithm with Weighted Stresses.

The incorporation of metrical stress into the difference measure is achieved by assigning differential weights to notes that start at different places in a bar. These can be shown as a weight map which, in the case of a double jig, could be as in Table 7.7.

Distance in Bar	Weight
0	4
1/8	2
2/8	2
3/8	3
4/8	2
5/8	2
otherwise	1

Table 7.7 Stress weights for 6/8 time.

The corresponding formula is

$$\textbf{Difference} = \sum_{k=1}^{n} |p_{1k} - p_{2k}| \, w_k ws_k$$

Where ws_k is the appropriate weight corresponding the start of window **k**.

The difference matrix produced when this algorithm is run using the set of weights in Table 7.7 is given in Table 7.8.

	a	b	c	d
b	**446**			
c	**83**	**363**		
d	**342**	**221**	**246**	
e	**475**	**54**	**370**	**208**

Table 7.8 Window and stress weighted melodic difference results.

Note (1) in this case, we have an even greater problem in that **a** and **c** are evaluated as being more distant that **b** and **e**. This is partially because we fail to take account of transpositions,

(2) by choosing some of the weights to be zero we can use the algorithm to select only notes at particular metrical positions and to exclude all others. For example if only weights at positions 0/6 and 4/6 have non zero values, the algorithms processes the accented tones of chapter 6.

7.3.5 Melodic Difference Algorithms Combined.

In the last section the different weights were combined by multiplying them together.

$$\textbf{Difference} = \sum_{k=1}^{n} |p_{1k} - p_{2k}| \, w_k \, ws_k$$

Where

w_k is the width of window **k**.

ws_k is the weight derived from metrical stress for window **k**.

The above formula can be expressed as

$$\textbf{Difference} = \sum_{k=1}^{n} |p_{1k} - p_{2k}| W_k, \quad \text{where } W_k = w_k.ws_k$$

7.3.6 Key/Transposition Independent Algorithm.

The question of creating a **key independent** version of the algorithm might be derived from a process of transposing one of the segments so as to minimise the difference.

From considering various transposed versions of one of these tune segments, such as where the second tune segment has been transposed up **m** semitones, we get

$$\textbf{Difference} = \sum_{k=1}^{n} |p_{1k} - p_{2k} - m| W_k$$

One possible way in which we can visualise a key-independent comparison being made is as a process of making multiple estimates of the distance by means of one of the previous algorithms, where we allow one of the tune segments to be transposed to all possible keys in the vicinity of the other segment. A difference is calculated for each key. We can illustrate this as follows, by considering a comparison to the following two related bar segments from no.61, "The Humours of Whiskey" from TDMOI.

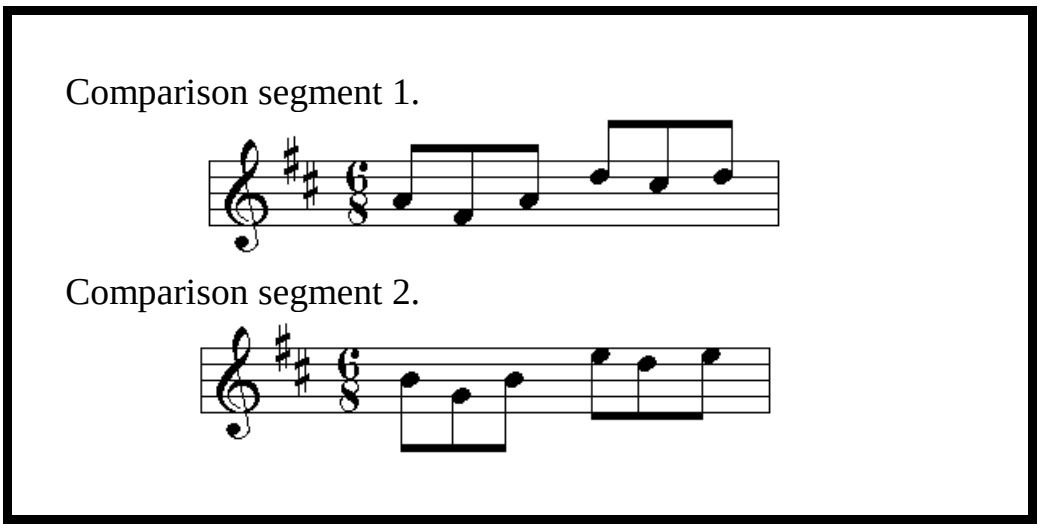


Fig.7.5 Two related tune segments from No. 61 in TDMOI for comparison.

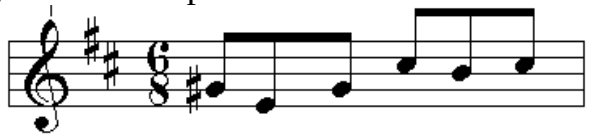
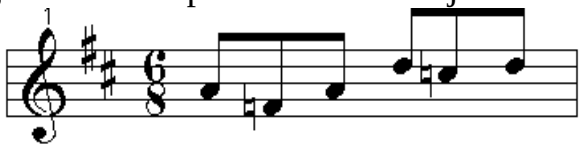
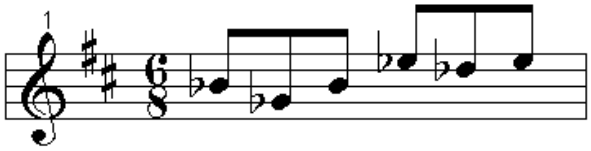
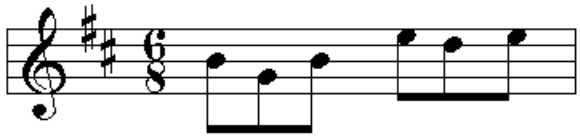
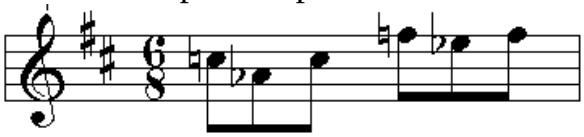
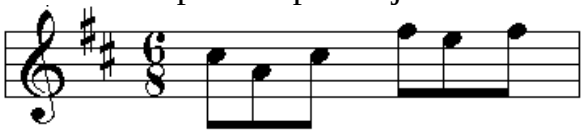
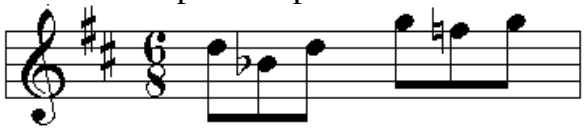
	Difference
Segment 2 transposed down a minor third. 	138
Segment 2 transposed down a major second. 	63
Segment 2 transposed down a minor second. 	63
Segment 2 at pitch. 	88
Segment 2 transposed up a minor second. 	163
Segment 2 transposed up a major second. 	238
Segment 2 transposed up a minor third. 	313.

Fig.7.6 Calculation of a transformationally independent difference.
The example illustrates the evaluation of difference between segment 1 and various transpositions of segment 2.

We can see that the difference calculation for the original untransposed method gives 88, but that if the second segment is transposed down either a major or a minor second, a smaller value of 63 results. The process of finding this difference is equivalent to finding the value of **m** which minimises

$$\textbf{Difference} = \sum_{k=1}^{n} |p_{1k} - p_{2k} - m| W_k$$

A well known theorem in statistics[118] enables us to find the required value of **m** which minimises the sum, without the repeated calculations involved above. **m** is the median value of the sequence of pitch differences, ($p_{1k} - p_{2k}$), with weight W_k associated with each difference. In statistics applications W_k is normally interpreted as a frequency. The use of this theorem gives us a way of arriving at the answer efficiently.

The following difference matrix was produced by a transposition independent difference algorithm using windows and stress weighting -

	a	b	c	d
b	**217**			
c	**42**	**200**		
d	**133**	**154**	**133**	
e	**188**	**63**	**170**	**92**

Table 7.9 Differences weighted by windows, stresses with transpositions.

If we use stress - note duration weights, where the duration of any note is taken as being at its onset, we get

	a	b	c	d
b	**433**			
c	**82**	**400**		
d	**317**	**292**	**342**	
e	**342**	**108**	**350**	**167**

Table 7.10 Differences weighted by durations, stresses with transpositions.

[118] A, C. Aitken, Statistical Mathematics, volume 1 (Edinburgh: Oliver and Boyd 1939), p.32.

We can see that the use of transpositionally independent comparisons have resulted in the **a-c** relationship being closer than the **b-e** one.

A program that incorporates all of these algorithms is given in Fig.7.7(c). The algorithm works for either a contour comparison or for a pitch difference comparison. The algorithm is written as a function which takes three parameters, two score iterators, representing the start of the two monophonic scores segments under comparison, and a rational length argument which specifies the time span over which the comparison is to be made. The algorithm returns a value, which gives an estimate of the melodic distance between the two segments. The work associated with windowing, such as the calculation of the window length, and the automatic stepping of the score iterators to the start of the windows, is achieved by the **traverse** function which is documented in Appendix 1. Different factors may be taken into account by setting switches, one of which selects contour processing. Alternately, various combinations of (1) note durations, (2) window durations, (3) stresses and (4) transposition processing may be selected.

```
float Stresses::getStressWeight(ScoreIterator & si)
{
 long tsn = si.getTimeSigNumerator();
 long tsd = si.getTimeSigDenominator();
 if ( (int)stressWeights[0] != tsn )makeStressVector(tsn);

 float returnWeight = 1.0;

 for (int count = 0; count < tsn; count++)
  if ( Rat(count, tsd) == si.barDist())
   returnWeight = stressWeights[count+1]+1.0;
 return returnWeight / stressWeights[0];
}
```

Fig.7.7(a) Calculation of stress weights.

```
float slopeWeight(int oldPitch1, int newPitch1, int oldPitch2,
                  int newPitch2)
{
 static float slopeWeight[] =
      { 0.0,   // same direction
        2.0,   // contrary motion
        1.0,   // one stationary, one moving
        0.0,   // both stationary
        0.0 }; // undefined
 int slopeIndex;
 if ( oldPitch1 == 0 || oldPitch2 == 0 ) slopeIndex = 4; // undefined
 else if ( (oldPitch1 > newPitch1 && oldPitch2 > newPitch2) ||
           (oldPitch1 < newPitch1 && oldPitch2 < newPitch2))
      slopeIndex = 0;                                  // similar motion
 else if ( (oldPitch1 > newPitch1 && oldPitch2 < newPitch2) ||
           (oldPitch1 < newPitch1 && oldPitch2 > newPitch2))
      slopeIndex = 1;                                  // contrary motion
 else if ( oldPitch1 == newPitch1 &&  oldPitch2 == newPitch2)
      slopeIndex = 3;                                  // both stationary
 else slopeIndex = 2;                   // one stationary, one moving
 return slopeWeight[slopeIndex];
}
```

Fig.7.7(b) Calculation of slope weights.

```
float difference ( ScoreIterator &si1, ScoreIterator &si2, Rat ln)
{

 float diffresult = 0.0;
 Rat toProcess = ln;
 int more = TRUE;
 Rat window = Rat(0,1);
 const int MAXNOTES = 1000;
 int noteCount = 0;
 transposeDist = 0;

 //                                for slopes comparison

 if (isSlopesSet())
 {
  int oldPitch1 = 0, oldPitch2 = 0;
  float x = 0.0;

  while (more)
  {
   traverse(si1, si2, window);
   if ( isDiatonicSet())
   {
    x +=  slopeWeight(oldPitch1, si1.getPitch7(), oldPitch2,
                                 si2.getPitch7());
    oldPitch1 = si1.getPitch7();
    oldPitch2 = si2.getPitch7();
   }
   else
   {
    x +=  slopeWeight(oldPitch1, si1.getPitch12(), oldPitch2,
                         si2.getPitch12());
    oldPitch1 = si1.getPitch12();
    oldPitch2 = si2.getPitch12();
   }
   toProcess = toProcess - window;
   if ( toProcess <= Rat(0,1)) more = FALSE;
   if ( si1.isLast() || si2.isLast()) more = FALSE;
  }
  return x/float(ln);
 }
 //                                  end of slopes comparison


 //               here we need to store pitch and weight information

 int * noteAr;
 float * weightAr;
 noteAr = new int[MAXNOTES];
 weightAr = new float[MAXNOTES];

 for ( int i  = 0; i < MAXNOTES; i++) weightAr[i] = 1.0;

 Stresses stress(si1.getTimeSigNumerator());
 while (more)
 {
  traverse(si1, si2, window);
  if ( window > toProcess )
       window = toProcess; // clip window if it exceeds range
  if (si1.getTag() == NOTE && si2.getTag() == NOTE)
  {
   noteAr[noteCount] = si1.getPitch12() - si2.getPitch12();
```

```
// calculate and add in stress weights

   if ( isDurationsSet() )
   {
    weightAr[noteCount] = 0;
    if ( si1.getRDuration() == si1.getRemainder() )
     weightAr[noteCount] += double(si1.getRDuration());

    if ( si2.getRDuration() == si2.getRemainder() )
     weightAr[noteCount] += double( si2.getRDuration());
   }

// window length weighting

   if ( isWindowsSet() )
    weightAr[noteCount] *=
                          float(window.numer())/float(window.denom());

// metrical stress weighting

   if (isStressesSet()) weightAr[noteCount] *=
    stress.getStressWeight(si1) + stress.getStressWeight(si2);
   noteCount++;
  }
  toProcess = toProcess - window;
  if ( toProcess <= Rat(0,1)) more = FALSE;
  if ( si1.isLast() || si2.isLast()) more = FALSE;
 }

 // we now get the median of the pitches, if appropriate

 int medianPitch = 0;
 diffresult = 0.0;
 if ( isTransposeSet())
  transposeDist = medianPitch = median(noteAr, weightAr, noteCount);
 for (i = 0; i <  noteCount; i++ )
 {
  diffresult += noteAr[i] > medianPitch ?
   (noteAr[i] - medianPitch)*weightAr[i]:
   (medianPitch - noteAr[i])*weightAr[i];
 }

 delete [] noteAr;
 delete [] weightAr;

 diffresult *= 100.0;     // scale up to make more readable
 return diffresult/float(ln);
}
```

Fig 7.7(c) General difference program.

7.3.7 Critical Value.

Before leaving the design of difference algorithms, a few points must be made. First there is a matter of arriving at a critical value in the algorithms so that they produce the best results possible. Let us assume here that we want to calculate a yes/no answer to the question, "are two melodic segments related or not." In the rather crude way in which these algorithms work, we have to find some number, called a critical value for the dividing line between 'similarity' and 'dissimilarity'. Critical values should divide unrelated segments which, ideally, should have a calculated difference greater than the critical value. Similar segments give a calculated difference less than or equal to the critical value. How effective such an artificial dividing line might be depends on the nature of the music under study and on the effectiveness of the difference algorithm. Procedures for estimating this dividing line or critical value can be manual or automatic. Manual estimation involves examining a sample corpus and classifying pairs of segments as being either similar or dissimilar. This is followed by running the difference algorithm and then by manually comparing the calculated difference values with our expectations (the 'difference' function is implemented in this study in such a way that by setting a software switch we can get the algorithm to output the values it calculates to a file). Assuming that we are dealing with simple melodic relations, we should be able to identify visually, a critical value that will work in most, if not all cases, especially if our goal of 'melodic similarity' is limited to exact or very close variants of the melodic segment. In some applications, there appears to be a clear cut numerical difference in segments that are similar from those that are not. In the use of the **diff1** function in the **PartsExpert** class, for example, it was possible to pick the value of 300 which worked well for double jigs. The distribution of the values produced by **diff1** in this case was strongly bi-modal and it was found that 300 divided it in two, in a satisfactory way.

7.3.8 Tuning of Melodic Difference Algorithms.

For the more general algorithms various weights are used, such as those involved in stress weights, for which values also have to be estimated. To do this we run our algorithm on a sample set of tunes or tune segments and examine the results produced. We then adjust weights by 'training' the difference algorithms on sample material. In some cases, from inspecting the results, we see instances where the algorithm produced the wrong result. Next we see if by adjusting some of the weights, we can eliminate this problem. We may find, for example, that by giving a higher relative weighting to metrical stresses, we may be able to solve a particular problem of mismatching. We follow this with re-running the algorithm, and

readjusting the critical value until we reach the best result. The process here is of a hit-and-miss nature. Often the adjustments which are made to solve one problem, result in introducing new ones. There is no guarantee, that this process will necessarily converge and result in a better algorithm.

There is much scope for further work. The use of mathematical optimising techniques is likely to prove fruitful here.

7.3.9 Segmentation for Melodic Difference Algorithms.

The current study uses an extremely primitive segmentation strategy which works reasonably well because of the regularities of the dance music.

7.3.10 Further Development of Melodic Difference Algorithms.

Difference algorithms in themselves are rather artificial constructs. In the final difference algorithm presented above, the various weights were combined in a multiplicative way and the results were added together. Further work is required to consider alternate ways of combining these. We could use addition or root-mean-square values, for example. Alternate approaches are possible in dealing with pitch, where it might be appropriate to use diatonic pitch numbers rather than chromatic pitch numbers in the calculations. Some approaches to these problems are outlined in 8.2.2.

7.4 Application of a Difference Algorithm to the Analysis of Form.

Using a difference algorithm, we can give a bar-by-bar analysis of a piece. This is done by labelling the first bar 'a', and then comparing bar 1 to every other bar in the piece. If any bar is sufficiently similar, we also label it 'a'.

Next we leave bar 1, and move to the next unlabelled bar, which we label as 'b'. We then compare this bar with all subsequent unlabelled bars, and label as 'b' all of those that are sufficiently close to the 'b' comparison bar.

We continue in this way, labelling the first still unlabelled bar as 'c' and complete similar processes to above. We continue for 'd', 'e', etc. until we have no unlabelled bar. An algorithm to do this analysis is given in Fig.7.8. This is followed by a listing of the calculated forms for CRNH1 in Table 7.11.

```
void form( String & str, Score &s, Rat In,
 float(*difference)( ScoreIterator &si1, ScoreIterator &si2, Rat ln),
 float criticalValue, int lid)

//form calculates the form of s, using the difference algorithm
//  difference for windows of length In
{
 ScoreIterator si1(s, lid), si2(s, lid);
 str = String();
 si1.locate();
 si2 = si1;
 int more1 = TRUE;
 int letterCount = -1;
 int countFirst = 0, countAhead = 0;
 for ( int count = 0 ; count < MAXCSLEN; count++) str[count] = 0;

 while ( more1 )
 {
  countAhead = countFirst + 1;
  if ( !si1.locate(BAR, countFirst+1)) more1 = FALSE;   //  end up
  else
  {
   // give label for next section
   if ( str[countFirst] == 0)
   {
    str[countFirst] = letterCount > 25 ?
                           'A' + ++letterCount:
                           'a' + ++letterCount;
    int more2 = TRUE;

    // search ahead and label all entries that match
    while (more1 && more2)
    {
     if ( ! si2.locate(BAR, countAhead+1)) more2 = FALSE;
     else if ( !si1.locate(BAR, countFirst+1)) more1 = FALSE;
     if (si2.isLast() ) more2 = FALSE;
     if (si1.isLast() ) more1 = FALSE;

     if ( more1 && more2 )
     {
      if ( str[countAhead] == 0 )
      {
       float x = difference(si1, si2, In);
       if ( x < criticalValue) str[countAhead] = str[countFirst];
      }
      countAhead++;
     }
     else if ( !more1 && !more2) str[countFirst] = 0;
                                              // unset prior allocation
    }
   }
  }
  if (si1.isLast() ) more1 = FALSE;
  countFirst++;
 }
 // put in the last part, if not already done
 if ( str[countFirst] == 0) str[countFirst] = 'a' + ++letterCount;
}
```

Fig.7.8 Program of algorithm for the calculation of forms.

```
Calculation of forms for file =\mdb\crnh1\djig.dir
Key transitions processed
Stresses processed
Critical Value = 40

1 Cailleach an Tu/irne                          abcd aefg hiij hifb klmd klne opqj hifb
2 Ple/ara/ca na Ce/ise                          abcd abce fgcd fgch ficb fice
3 Carraig an tSoip                              abcd abce fghd fghi
4 Pingmeacha Rua agus Pra/s                     aabc adef ghij klmf nonl pqef
5 Gleanta/n na Samhairci/ni/                    abcd abce fghb ijjb
6 Tolladh an Leathair                           abcd aefg abcd aefh fifh fjkl
7 An Fhuiseog ar an Tra/                        abcd aefg hfhd hfdd
8 Bruacha Thalamh an E/isc                      abcd abce fghd fghe fghd abce
9 Cathaoir an Phi/obaire                        abcd abce fghi jkle
10 Ballai/ Lios Chearbhaill                     abcd abef cdcg cdef cdhi jkea
11 Port Ui/ Cheallaigh                          abcd abef ghgi ghjf
12 Port Liadroma                                abac adef gdgf ghif
13 An Maide Draighin                            abca abcd abca abce fgca fgce hgca abcd
14 Buachcilli/ Bhaile Mhic Annda/in             abcd ebfg hijd hicc
15 An Boc sa gCoill                             abac ddec fbgh fbec abic abjc
16 Sean-Tiobrad A/rann                          abac abde fghi djkl
17 Bi/mi/d ag o/l is ag po/gadh na mBan         abac adef ghgi gjkf
18 Ard an Bho/thair                             abcd aefg abcd aefh ijck ijlm
19 I/oc an Reicnea/il                           abac adec fghc ijec
20 An Buachailli/n Ba/n                         abcd abce fgcd fgce fgcd abce
21 Port an Bhra/thar                            abcd efgh ijik iglh
22 Port Shean tSea/in                           abcd abef abcd abec ghgi ghec
23 Scaip an Puiteach                            abac adef gcgh gief
24 An Pi/osa Deich bPi/ngne                     abcd cbce fgfh ijkd
25 Luighseach Nic Cionnaith                     abcd abef ddgg ddgf
26 Droim Chonga                                 abac abdc efeg edbc
27 An Buachailli/n Bui/                         abac adef ghef gijk lalk lmnf opok opnf
28 Na Ge/abha sa bPortach                       abac adef abac adeg hijk jleg jmjk jmnk opoq oreg stuq sveg
29 An Gaoth Aniar Andeas                        abcd efgh ijij iklh
30 An Gandal i bPoll na bhFatai/                abac abdc efeg ehdc
31 Banri/on na Luacra                           abac dbef abac dbeg hiji hieg hiji hief kklm kked
32 Ma/irsea/l na nIoma/naithe                   abcd aefg abcd aefh gigj klfm nnop nnqr sagt sufh sagt sufg
```

33 An Bo/thar Mo/r go Sligeach	abcd abef ghij ghkl mhin okpl
34 Spara/n Airgid na Cailli/	abcd abef abcd abeg hicd jikg hicd jikf
35 Port an Riaga/naigh	aabc aade fgbh fgde
36 An Ceolto/ir Fa/nach	abac abde fgfh fgij klmn klfe
37 An Ro/s sa bhFraoch	abcd abef ghij gklm ghij gkld
38 Ruaig an Mi/-a/dh	abcd efag chch ehig jklm nfig
39 Airgead Re/alach	abcd ebcf ghgi ghjf
40 An Pi/opa ar an mBaic	abcd eefg bbhi bbij bbhi keld
41 Cailleach an Airgid	abac abde fbfg fhie
42 Gearrchaile Bhaile Ui/ bhFiacha/in	abcd abef ghgi jbef
43 Siamsa Mhuilte Faranna/in	abbc abde fggg fghe
44 Pa/draic Mac Giollarna/th	abcc aded abcc aded fghi fhed
45 Port Ti/neatha	abcd abce fgcd fgce
46 Gearo/id de Barra	abcd efgh abij efgk ilmn iogp ilmn iqgh
47 Fa/inne O/ir Ort	abcd ebcf eghd egij klmn klmo pqrs pqrf
48 Rogha Liadroma	abac abde bfcg bhig
49 Port Ui/ Fhaola/in	abac adec fghi fgjk
50 Port Ui/ Mhuirgheasa	abcd abef ghgi ghjk ghgl mnok
51 Port Shligigh	abcd abef ghcd ghef
52 An Cru/ Capaill	abcd ebfg hijg akfg
53 An La/ i ndiaidh an Aonaigh	abcd efgb hbhd ifgj
54 An Seanchai/ Muimhneach	abac abde fgac fgde

Table 7.11 Forms in CRNH1.

7.5 Hierarchical Possibilities of Building more Complex Software.

What we have done is to take relatively simple tasks and to build hierarchically so that we achieve more complex algorithms by assembling aggregates from simpler components. All of this is in turn built upon the score abstraction. The **form** function itself is built using the difference algorithm, which in turn uses the **traverse** algorithm. None of these individual algorithms are complex. However, by hierarchically decomposing, or by synthesising, we can assemble increasingly complex problems from more simple ones. In the next example, we will now go one step further and use the 'form' function together with the **Store** class for summarising information about the form of tunes in both collections.

7.6 Frequency Distributions of Forms.

The algorithm in Fig.7.9 uses the form calculating algorithm, but instead of printing out individual forms, it calculates the frequency distribution of the form in the tune part of a double jig.

```
Store<String> store(100);

while (getNextScoreNames( argv[argc-1], fname))
{
 Score s1(fname);
 String str;

 form( str, s1, s1.getTimeSig(), difference, criticalValue);

 str.sub(0,7);
 store.put(str);
}
```

Fig.7.9 Program for forms frequency distribution.

```
Calculation of forms for file =d:\mdb\crnh1\djig.dir
Key transitions processed
Stresses processed
Critical Value = 40


Form        Frequency

abcd efgh   3
abcd efgb   1
abcd efag   1
abcd eefg   1
abcd ebfg   2
abcd ebcf   2
abcd cbce   1
abcd aefg   5
abcd abef   10
abcd abce   7
abcc aded   1
abca abcd   1
abbc abde   1
abac ddec   1
abac dbef   1
abac adef   5
abac adec   2
abac abde   5
abac abdc   2
aabc adef   1
aabc aade   1
```

Table 7.12 Frequency distribution of form for the tune part of double jigs in CRNH1.

As can be seen from these examples the most commonly occurring forms in CRNH1 in Table 7.12 are also the most commonly occurring ones in TDMOI. This is shown in table A3.3 in appendix 3.

7.7 A Compute-Intensive Task.

Suppose we ask the question "How many distinct tunes are there in the collection?". The task of defining what we mean by a distinct tune needs consideration. How do we compare two-part tunes with three-part tunes? How do we handle the case where in one part of the collection, we have a tune that reappears elsewhere but with the tune and the turn part in reverse order? In this section we will ask a simpler question. We will develop an algorithm to list all pairs of tunes with one or more similar 8-bar segments in the corpus. The computer will compare part 1 of tune no.1 with all other 8-bar segments in the collection (365 tunes with two or more 8-bar segments in each one in the case of O'Neill's). It will then move onto part 2 of tune no.1 and make a similar comparisons. We continue in this way until all 8-bar segments of tune no. 1 has been compared with all

other 8-bar segments of other tunes in the corpus. Likewise, when all comparisons of segments of tune 1 have been completed, we then process all 8-bar segments of tunes 2, 3, and so on until all 8-bar segments in the corpus have been compared. This is repeated with the next tune, and so on until every 8-bar segment of each tune is compared with every other such segment. A program to do this processing is given below in Fig.7.10.

```
int countOuter = 0;
int countInner = 0;

while( getNextScoreNames( String(argv[argc-2]), fname1,1))
{
 countOuter++;
 Score s1(fname1);
 ScoreIterator si1(s1);
 int firstscore1 = TRUE;
 countInner = 0;

 while (getNextScoreNames(argv[argc-1], fname2))
 {
  countInner++;
  if ( !identicalContents || countInner > countOuter )
  {
   Score s2(fname2);
   ScoreIterator si2(s2);
   int firstscore2 = TRUE;
   int count1 = 0;
   int count2 = 0;
   si1.locate();
   si2.locate();
   Rat segLength = s1.getTimeSig()*(Rat(nbars,1));
   int more1 = TRUE;
   while ( more1 )
   {
    count2 =  0;
    if ( !si1.locate(BAR, count1*nbars+1)) more1 = FALSE;
    else
    {
     int more2 = TRUE;

     while (more1 && more2)
     {
      if ( ! si2.locate(BAR, count2*nbars+1)) more2 = FALSE;
      else if ( !si1.locate(BAR, count1*nbars+1)) more1 = FALSE;
      if (si2.isLast() ) more2 = FALSE;
      if (si1.isLast() ) more1 = FALSE;

      if ( more1 && more2 )
      {
       float x = difference(si1, si2, segLength);
       countComparisons++;
       if ( x < crit )
       {
      if (firstscore1)
      {
       fout  << "\n(" << si1.getString(NUMBER) << ")"
```

```
              <<  si1.getString(TITLE) << " - "
              << si1.getString(ETITL);
       firstscore1 = FALSE;
      }
      if (firstscore2)
      {
       fout  << "\n    (" << si2.getString(NUMBER) << ")"
             << si2.getString(TITLE) << " - "
             <<  si2.getString(ETITL);
       firstscore2 = FALSE;
      }
      fout  << "\t" << count1+1 << "="
            << count2+1 << " (" << setprecision(1) << x;
      if ( isTransposeSet() ) fout << ':' << getTransposeDist();
      fout << ") ";
       }
       count2++;
      }
     }
    }
    if (si1.isLast() ) more1 = FALSE;
    count1++;
   }
  }
 }
}
```

Fig.7.10 Program of algorithm for exhaustive search using fixed length similar segments.

When this algorithm is run on a single file of size n, the time taken is $O(n^2)$. For bigger corpora this can be expected to take disproportionately longer to run than for smaller corpora. For example, if it takes 10 seconds to compare all 8-bar segments in two tunes, then a corpus of 100 tunes of the same length will be processed in 99x98x10 = 97020 seconds or just over 26 minutes. A corpus of 1000 tunes would take 999x888x10 = 8871120 seconds or over 102 days! A corpus of 419 jig tunes will take 418x417x10 = or over 20 days. The assumption of 10 seconds processing per tune is roughly what can be achieved with a 20MHz 386 PC. Fortunately, machines that are faster by over an order of magnitude are now commonplace, and the processing time for 365 tune of TDMOI can be reduced to a matter of hours .

7.7.1 Comments.

This kind of comparison does an enormous amount of computing. The current applications show which parts of tunes are related, within the limitations of the difference algorithm and critical value used. When the algorithm is run for the systematically organised Breathnach collection, it produces little output as, by means of a card index, Breathnach eliminated duplicates and closely related tunes, see Table 7.13. The

difference algorithm used here employs metrical stress weights, window weights and the transposition algorithm with a critical value of 100. The pair of tunes identified as being related in relation to their first parts is acknowledged as such in the notes on the tunes given by Breathnach in CRNH1.[119] The difference result in this case was 98.6 which was just inside the critical value of 100. The difference was taken using a transposition of 0 semitones.

```
Calculation of distances for files =d:\mdb\crnh1\djig.dir and
                                   =d:\mdb\crnh1\djig1.dir
Key transitions processed
Stresses processed
Window widths processed

(3)Carraig an tSoip -
    (14)Buachcilli/ Bhaile Mhic Annda/in -      1=1 (98.6:0)

54 items processed from file =d:\mdb\crnh1\djig1.dir
54 items processed from file =d:\mdb\crnh1\djig.dir
10578 comparisons made

critical value =100
```

Table 7.13 Result of exhaustive search of CRNH1.

The situation for O'Neills is far less structured, as shown in table A3.4 of appendix 3. This is not solely due to O'Neill's lack of appropriate tools, it is also part of the nature of material in an oral tradition.

In addition to using this program for searching through a collection for duplicates or closely related tunes, this program can also be used to identify duplicates or closely related tunes across two collections. Table A3.5, in appendix 3, shows some interrelationships between TDMOI and CRNH1.

[119] Breandan Breathnach, op.cit. 1963, pp.87-88.

Chapter 8. Achievements, Further Work and Conclusions.

8.1 Achievements.

8.1.1 The creation of a score representation in accordance with principles of informational completeness, objectivity, extendibility and abstraction.

8.1.2 The modelling of a polyphonic score which unifies the physical score with its computer representation. This is achieved by the following mappings and structures.

- Conceiving of the score as a container of various entities.
- Mapping relationships of vertical contiguity to simultaneity.
- Mapping relationships of horizontal contiguity to an absolute score time scale.
- Representing an absolute score time as a rational number displacement from the start of the score. Alternately it may be represented as a bar count and a rational displacement within a bar.
- The representation and automatic resolution of scoping information, such as occurs in time signatures, slurs and accidental alterations.
- The availability of methods to construct and edit a score.
- The availability of methods to retrieve any information present in a score.
- A mechanism of a score iterator for navigating through a score. A number of basic score iterators are developed within the system.
- The facility for constructing user-defined score iterators.

- The identification of the sense of line by means of native and of user written score iterators.

8.1.3 The environment in which the score is represented has the following desirable characteristics.

- A high level of complexity hiding is involved.
- Algorithms of arbitrary complexity may be expressed in it.
- The user interface is simplified by means of the use of polymorphism.
- Extendibility is facilitated through the mechanism of inheritance.
- The environment has the potential to act as a repository for new components, and hence facilitates the building of algorithms of arbitrary complexity by means of software reuse.

8.1.4. Methodologies are proposed and demonstrated which have the following characteristics.

- Hypotheses may be formulated in a highly structured manner by expressing them as algorithms. Such structuring necessitates the conscious resolution of ambiguities and forces the musicologist into a precise statement of the task environment. This process may result in drawing attention to ambiguities and to inconsistencies in the original hypothesis formulation.
- There is a focusing on the corpus as the 'evidence' for proving hypotheses.
- The testing of hypotheses against the corpus is carried out in an objective manner which has parallels with scientific method.
- Analytic methods may be combined to form more complex analyses.
- The otherwise limited capabilities of musicologists, because of lack of time, energy, attentiveness, and accuracy, is greatly extended.

8.1.5 The extendibility of the environment is demonstrated by the construction of new components. These include classes for pitch class sets, non inversionally equivalent pitch class sets, pitch tuples and a parts expert. Use of these classes is demonstrated in chapters 6 and 7.

8.2 Proposals for Further Work.

8.2.1 Development of the Basic Level of scoreView.

Currently **scoreView** is designed for representing a large subset of all scores written in common practice notation. Individual new score features, such as special symbols, which are not currently part of **scoreView** can readily be added within the code of **scoreView** as the need arises, using existing mechanisms.

One area for augmentation of **scoreView** is in extending its capacity for processing general polyphony. Currently, the representational capability of **scoreView** includes multi-stave scores. A limitation arises where polyphony is used on single staves. Each stave can accommodate rhythmically dependent polyphony only. That is polyphony, in which concurrent notes and rests share the same time values. This limitation makes the representation of most piano music infeasible at present, but allows for the representation of most choral and orchestral music. The overcoming of this limitation requires some modest augmentation of the internal representations in **scoreView**, the enhancement of the input translator, and extensive testing with polyphonic corpora.

Another target area for augmentation of **scoreView** is in the provision of new iterators that could be of use in the handling of polyphonic music. The available polyphonic iterator traverses the score in standard traversal order. It is inevitable that some kind of parallel traversal of all the simultaneous notes in a score will be found desirable in harmonic studies. The basic mechanism for doing this is already in place in the internals of the **traverse** function, which operates on two melodic lines only. The development of such score iterators, however, is best left to special projects. Two other iterators that should prove useful are (1) an iterator for tracking divisi lines; and (2) an iterator which follows two implied lines of polyphony. This phenomenon is referred to

as the 'streaming effect'. It is documented by Bregman[120] along with general issues of sequential integration.

The development of input translators for a range of sources is highly desirable, given the volume and diversity of machine encoded scores that are available. Target encoding schemes for these include various dialects of DARMS, as well as for Score and Finale code.

The development of a graphical component within **scoreView** for representing score information on a VDU and the provision of an interactive editor and input system are of interest.

8.2.2 Development of Basic Tools with scoreView.

The crude melodic difference algorithms in chapter 7 stand to benefit from a number of refinements. Firstly a number of mathematical optimisation techniques could be applied to adjust the weights used for metrical stresses. Such techniques would enable the weights to be arrived at by training the system using a sample corpus in which relationships between the tune segments have been classified in advance. Secondly a closely allied problem, that of identifying phrases, could be tackled. Phrase identification is a good target task for algorithmic implementation. Some approaches to this problem can be found in the work of Narmour, Lerdahl and Jackendoff which is discussed in sections 8.3.2 and 8.3.3. The problem of phrase identification in the case of monophonic folk melodies has been tackled by Ahlbach.[121]

Another kind of algorithm that is of great use in corpus analysis is one which identifies the mode or tonal centre of a piece of music. As so much tonal music theory depends on knowing the tonal centre in advance, the development of such algorithms is essential for initial investigations in many areas of corpus musicology. There are

---

[120] Albert S. Bregman Auditory Scene Analysis (Cambridge, Massachusetts:The MIT Press 1990), pp.47-211.

[121] Sven Ahlback, "A Computer-Aided Method of Analysis of Phrase Structure in Monophonic Melodies" Irene Deliege Procedings of the International Conference for Music Perception and Cognition (Liege 1994), pp.251-2.

numerous studies that can be drawn on in designing such algorithms, including the work of Longuet-Higgins and Steedman[122], and Krumhansl[123].

Another area of endeavour is the integration of **scoreView** into a database for information retrieval applications. This involves the development of strategies for rapid searching and retrieval of information from large databases, and in particular for the retrieval of information using music as part of a search criterion. In addition to having the database handle score information, it is desirable to be able to deal with other kinds of information. Such information might be textual, sonic and pictorial in nature, and should include possibilities for the representation of music analyses.

The development of a more limited, but easy to use and reasonably powerful tools on top of **scoreView** is another possibility. Page's[124] system is an example. It embodies a search tool which could be usefully implemented in **scoreView**.

8.3 Use of scoreView in Research.

The **scoreView** environment is open. It does not force the user into any special area of research. It can be characterised as 'a potential solution looking for a problem'. Imagination is the main limitation in its use.

There are however, a number of existing areas of activity in which **scoreView** should prove particularly useful. Some of these are listed in this section. Corpus-based musicology is relevant to all of them in that it could provide a testing ground for models. This might be done by having **scoreView** used to build a simulated ‘performer’ and by constructing a cognitive model which ‘listens’ to the music. This does not necessarily mean that ‘performing’ is fully implemented as a performer model. The performer model needs be implemented only to the extent that is required by the ‘listening’ model.

---

[122] H. C. Longuet-Higgins and M. J. Steedman “On the Interpretation of Bach” in Machine Intelligence, volume 6 (1971), pp.221-41.

[123] Carol L. Krumhansl Cognitive Foundations of Musical Pitch (Oxford 1990), pp.77-110.

[124] Stephen Dowland Page, op.cit.

8.3.1 Psychomusicology.

In a number of publications[125]Otto Laske conceptualises musicology as a science comprising three subdisciplines: music analysis, psychomusicology, and sociomusicology. He introduces the notion of a process model of music. He says:

> "Such a model is a joint description of musical structures and of the mental processes required for their production, reproduction, and comprehension. More specifically a process model is a model of the process by which musical structures are actually generated by a musician through analysis, performances, improvisation, composition or listening. The process model has a threefold purpose. First it is a description of musical structures as they are held in human memory during some task performance. Second, the model describes the "performance program" (performance taken here in the sense of activity) a human musician needs to activate to manipulate musical structures held in memory. Third , the model serves to embed the first in the second description, thereby explicating musical structures in the medium of processes that generate them."[126]

Two such process models of music have been developed to the extent that part of them can be modelled on a computer. They are treated in the following sections.

8.3.2 Narmour's Implication Realisation Model.

The implication-realisation model was introduced by Eugene Narmour in 1977.[127] In recent years the first two volumes[128] of a planned four volume series have been published which greatly develop the model. The model is one of a listener which is based on Gestalt psychology. Narmour's primitives are the notes that form melody. He proposes the existence of an input system in a music listener, which operates on a bottom-up level. This assumes the existence of an innate syntactic parametric scale.

"As we shall see, a syntactic parametric scale is an automatic, "brute" input system that is domain specific, mandatorily operative, and computationally reflexive."[129]

---

[125] Otto E. Laske Music, Memory and Thought (Ann Arbour: UMI 1977), Otto E. Laske Psychomusicology (Bombay and Baroda: Indian Musicological Association 1985), Otto E. Laske "Introduction to Cognitive Musicology." Computer Music Journal volume12, no.1 (Spring 1988), pp.43-57.

[126] Otto Laske, op.cit., 1985, in preface (page unnumbered).

[127] Eugene Narmour Beyond Schenkerism (Chicago 1977).

[128] Eugene Narmour The Analysis and Cognition of Basic Melodic Structure (Chicago 1990); Eugene Narmour The Analysis and Cognition of Melodic Complexity (Chicago 1992).

[129] Eugene Narmour, op.cit., 1990, p.4.

The input system determines the degree of implication between patterns of similarity A + A -> A and differentiation A + B -> C, (where -> = implies), and also determines closural and non closural functions.

In melody, these elements A, B and C can stand for either intervallic patterns, or pitch elements.

Narmour forms five kinds of melodic archetypes:

1. Process or iteraton (A+A, nonclosural);
2. reversal (A+B, closural);
3. registral return;
4. dyad;
5. monad.

The syntactic parametric scale hypothesises that any pair of melodic pitches transmits separate intervallic and registral messages to the listener. Narmour hypothesises that three-note sequences give rise to exactly eight shapes. In classifying these he distinguishes small intervals such as thirds from large intervals such as sixths.

D = small interval to identical small interval, same registral direction;
P = small interval to similar small interval, same registral direction;
R = large interval to a smaller interval, different registral directions;
IP = small interval to similar small interval, different registral directions;
VP = small interval to large interval, same registral direction;
ID = small interval to same small interval, different registral directions;
IR = large interval to small interval, same registral direction;
VR = large interval to even larger interval, different registral directions.

Durational cumulation and/or metric emphasis parses these shapes into contiguous structures.

The above structures are bottom-up style shapes. Top-down style structure interacts with these ever-present style shapes. Narmour specifies various conditions for closure such as the effects of intervallic motion and of duration and of metrical emphasis.

The style shape constructs are eligible candidates for algorithmic expression. There exists a corpus to test the validity of these rules, in the form of some of the numerous examples given by Narmour in his recent pair of books. This endeavour, if successful, could be followed by attempts to model the syntactic parametric scale, and to model closure and non closure.

Narmour's approach has attracted some negative criticism from Stephen Smoliar.[130]

8.3.3 Lerdahl's and Jackendoff's Model.[131]

The Lerdahl and Jackendoff model has origins in Schenkerian analysis, linguistics and cognitive psychology, and in particular in Gestalt psychology. Its main focus is on modelling aspects of an idealised listener. It focuses on "those components of musical intuition that are hierarchical in nature".[132] It proposes techniques for analysing the grouping structure of a piece into a hierarchical segmentation consisting of motives, phrases and sections. Using time span reduction it organises pitches into a hierarchy of structural importance with respect to their position in grouping and in metrical structure. By means of prolongational reduction it assigns to the pitches, a hierarchy that expresses harmonic and melodic tension and relaxation, continuity and progression. Non hierarchical structures, such as those of instrumentation , timbre and dynamics are not formalised. It proposes methods of analysing music using a series of different types of rules. The first type of rule, well-formedness rules, represents possible structural descriptions. Preference rules, on the other hand, determine how selections might be made between conflicting rules. Transformational rules apply certain distortions to the otherwise strictly hierarchical structures provided by the well-formedness rules.

---

[130] Stephen W. Smoliar "The Analysis and Cognition of Basic Melodic Structures: The Implication-Realization Model by Eugene Narmour" (Review) in In Theory Only volume 12, nos.1-2 (1991), pp.43-56.

[131] Fred Lerdahl and Ray Jackendoff A Generative Theory of Tonal Music (Cambridge, Massachusetts, The MIT Press 1983). See also Nichola Dibben "The Cognitive Reality of Hierarchical Structure in Tonal and Atonal Music" in Music Perception volume 12, no.1 (1994), pp.1-25, and Lloyd Daws, John R. Platt and Ronald Racine "Inference of Metrical Structure from Perception of Iterative Pulses within Time Spans Defined by Chord Changes" in Music Perception volume 12, no.1 (19994), pp.57-76, and Irene Deliege "Grouping Condition in Listening to Music: An approach to Lerdahl and Jacklendoff's Grouping Preference Rules" in Music Perception volume 4, no.4 (1987), pp.325-360.

[132] Lerdahl and Jackendoff, op.cit., p.8.

A formal grammar is proposed for each component rule. Such rules cover the bottom-up aspects of the generative theory. Such generative rules have their analytic counterparts, which can be used in an analytic manner.

It is over a decade since the publication of this model. Some experimental evidence is emerging, most of which either supports or proposes modifications to aspects of the model. A series of reviews of the model have appeared in print within the last year under the collective title of "A Generative Theory of Tonal Music by Lerdahl and Jackendoff: 10 years on".[133] Many related articles have also appeared in Music Perception.

8.3.4 Biomusicology.

All of the previously described analytic methods depend on introspection to explain mental processes. There is no guarantee that such introspections accurately reflect the actual processes involved. Alternate approaches are available. One area in which progress is being made involves tackling the problem by looking at underlying biological systems and trying to explain mental processes in terms of biological processing. The areas in which this kind of exploration has had the greatest success is in studying peripheral mechanisms. In the case of hearing, quite a lot is known about the mechanisms that operate in the ear. Impressive models have been built.[134] Some of these model the kinds of processing that the ear does on sound. Hypotheses on plausible ways in which the brain might process pitch exist and can be used to explain some of the characteristics of hearing. There exists a plausible explanation of the processing of harmonic sound combinations.

The most comprehensive attempt to take an overview of the wider biological aspect of music is made by Wallin.[135] He views music as having

---

[133] Nattiez; E. Bigand, F. Lerdahl and M. Pineau; J. London; D. Bettrand; M Botello; P Halasz and D.R. Stammen and R. Pennycook in "A Generative Theory of Tonal Music by Lerdahl and Jackendoff: 10 years on" in Proceedings of the International Conference for Music Perception and Cognition (Liege 1994), pp.255-70.

[134] Richard F. Lyon “A Computational Model of Filtering, Detection and Compression in the Cochlea” in Preceedings of the IEEE International Conference in Acoustics, Speech and Signal Processing (Paris, France May 1982).

[135] Nils L. Wallin Biomusicology (Stuyvesant: Pendragon Press 1991).

"its primary base in man's biological inheritance, not in his cultural heritage."[136]

Wallin arrives at a draft definition of music as follows:

> "Music is an open system of evolving structures growing into sounding artefacts which not only consume actual time but also generate virtual time; the system and its space-time structures are ultimately conditioned by bio-geocultural parameters of behaviour and deportment. Music is basically perceived unilaterally in the right cerebral hemisphere through the auditory system in a bilateral co-ordination with senso-motoric limbic and associative brain functions (the autonomous system included) within a framework of multimodal experiences."[137]

Whereas complete biological processes are too complex to model, aspects of them have been tackled with some success. Connectionist models have been successful in building systems which mimic some aspects of human capacities in areas such as the perception of rhythm.[138] In particular those models which use neural networks have been found to be appropriate. Some proposals for such have been made by Smoliar.[139]

8.4 Conclusions.

This thesis demonstrates the feasibility of an environment for the development of corpus-based musicology. Such an environment can provide a rich source of new musicological possibilities. It creates the possibility of investigating uncharted territories in music theory. It offers possibilities for supporting sophisticated analytic and generative models. It provides a repository for the incremental building of future analytic systems of arbitrary complexity, limited only by the imagination of the user.

---

[136] Nils L. Wallin, ibid., p.xx.

[137] Nils L. Wallin, ibid., p.16.

[138] A cross section of connectionist approaches is found in Peter M. Todd and D. Gareth Loy Music and Connectionism (Cambridge, Massachusetts: The MIT Press 1991). See also Peter Desain and Henkjan Honing Music, Mind and Machine (Amsterdam: Thesis Publishers 1992).

[139] Stephen W. Smoliar "Elements of a Neuronal Model of Listening to Music" in In Theory Only volume 12, nos.3-4 (Feb 1992), pp.29-46.

Appendix 1 - scoreView User Manual.

scoreView Users Manual.

Conventions, Data Types and Classes of scoreView.

Conventions

Lowercase letters are not used in enumerated values and constant names such as TITLE, COMPOSER, TIME_SIG, MAXCSLEN and N8.

All names of classes, structs and unions start with an uppercase letter.

All variable and function names start with lowercase letters. Here a single uppercase letter indicates a separator in compound names, for example in **getString()**.

Constants.

The following constants are defined in score.h

The symbol TRUE is of type const int with the value 1.
The symbol FALSE is of type const int with the value 0.
The symbol NULL is of type const int with the value 0.

Analytic use: Classes representing objects in a score are accessible through class **ScoreIterator**. All such classes have a member function **getTag()** which returns an identifier for the object. This enables us to ask what kind of object is at the current position. **ScoreIterator** has two additional member functions, one of which, **getString()**, returns a textual version of the values associated with the object. **getName()** returns a string which is descriptive of the class. For example **si** is an object of class **ScoreIterator** which is currently pointing to a quaver middle C, the

si.getTag() returns the value NOTE
si.getName() returns the string "NOTE"
si.getString() return the value **"C5 [4].**

tagType cvtStrTag(const String & s); converts a string version of tag to a tagType.

It is anticipated that only a few of the classes in this chapter will be used by analysts. With the exception of analysts who are involved in generative studies, only two basic classes of **scoreView** will suffice for most processing. These are classes **Score** and **ScoreIterator**. Only a small set of member functions and operators of these classes are needed for analysis. These are the ones involved with extracting and processing information from the score representations. Such functions and operators are underlined in the manual.

Manual Pages for Classes Listed Alphabetically.

class Barline

Purpose: To represent barlines in a score.

Manager Functions and Operators

Barline(barType br = L, int br_n = 0);
Barline(const Barline & br);
virtual ~Barline();

where **barType** is one of
CLHLC, CLLC, CLH, HLC, CLL, LLC, CLC, SHORT, INVISIBLE, LL, CL, LC, H, L, DOTTED

Access Functions

The following function set and retrieve various fields in a Barline object.

void putBarNo (int bn);
Bar number is set to **bn**.
void putBar(barType br, int bn);
The bar type is set to **br** and the number to **bn**.
int getBarNo(void) const;
Returns the bar number.
barType getBarType(void) const;
Returns the bar type.
void clearAttributeSet(void);
Makes the attribute set of the barline null.
void putAttribute(const attrType & nr);
Puts the attribute **nr** into the attribute set of the barline.
void putAttributeSet(const Set & s);
Copies the set **s** into the attribute set of the barline.
Set getAttributeSet(void) const;
Returns a copy of the attribute set of the barline.

Valid attributes for a barline include **FERMATA, DA_CAPO, DA_CAPO_AL_SEGNO, DA_CAPO_AL_FINE, REPEAT1,** and **REPEAT2.**

String getString();

Returns one of the following. ":/I/:", "://:", ":/I", "I/:", ":://", "//:", ":/:", "[|]", "[/]", "//", ":/", "/:", "I", "/", "|"

These are the textual equivalents of the **barType** identifiers listed above.

String getName();

Returns the string "Barline".

class Clef

Purpose: To represent a clef in a score.

Manager Functions and Operators

Clef(clefType c = NOCLEF);
Clef(const Clef & cl) : Glue(cl), clef(cl.clef);
virtual ~Clef();

where **clefType** is
enum
clefType
{
FRENCH_VIOLIN, SOPRANO, MEZZO_SOPRANO, TREBLE, BASS,
ALTO, TENOR, BARITONE, NOCLEF
};

Access Functions

void putClef(clefType c);
Clef is set to c.
clefType getClef(void) const;
Return the clef value.
String getString();
returns one of
"FRENCH_VIOLIN", "SOPRANO", "MEZZO_SOPRANO", "TREBLE", "BASS",
"ALTO", "TENOR", "BARITONE".
These are the textual equivalents of the **clefType** identifiers listed above.

String getName();
Returns the string "Clef".

Comment

Clef has an open scoping mechanism.

class Duration

Purpose: To represent the duration abstraction of notes and rests.

Manager Functions and Operators

Duration(durType d = N4, int dot = 0);
Duration(const Duration & dr);
virtual ~Duration();
int operator==(const Duration & dr) const;

where **durType** is
enum durType { N0, N1, N2, N4, N8, N16, N32, N64, N128 }
Dot is the number of dots in the duration.

Access Functions

durType getHead(void) const;
Returns the duration of the object in whole notes, half notes, quarter notes, etc.
int getDots(void) const;
Returns the number of dots associated with the duration.
const Duration & getDuration(void) const;
Returns a copy of the Duration object.
void putHead(durType d, int dts = 0);
Sets the duration type and the number of dots in the object.
void putDots(int d);
Sets the number of dots for the duration.
String getString();
Returns a string description of the duration in the form of <ordinal durType value><number of dots>.
String getName();
Returns the string "Duration".

Implementer Functions

Rat getRDuration(void) const;

Returns the duration as a rational value.

Associated Functions:

Rat rDur(durType d, int dots = 0);

Converts a duration to a rational number.

Constants

A constant of type duration is declared with the name **DUMMYDURATION.**

class FrequencyStore

template <class T>
class FrequencyStore

Purpose: To store copies of objects in an ordered frequency distribution. Objects of class T must have the operators =, == and > defined, and must have a default constructor. The == operator should compare all relevant components of objects for equality. A function must be defined for objects of class **T** which is used to initialize the class of **T** and has the following signature.

void init(int);

Manager Functions and Operators

FrequencyStore(int storeSize = 0, int cellSize = 0);

Creates a store for **storeSize** objects. Each of the contained objects is initialized by calling its member function **init(cellSize)**.

~FrequencyStore();

void init (int storeSize, int cellSize);

Initializes a empty store to be of size **storeSize**, creates objects of type **T** for each element of the score, and invokes the **init(cellSize)** function for each of the created objects.

FrequencyStore & operator = (FrequencyStore<T> &t);

Access Functions

int isEmpty();

Returns **TRUE** if there are no members in the store.

void put(T tm);

Puts a copy of the object **tm** in the frequency store, if it is not already present. Increments the corresponding frequency. If the array is inadequate in size, its size is effectively doubled, and a message is sent to **cerr**.

T getValue(int i);

Gets the value of the i-th element from the frequency store.

int getFrequency(int i);

Gets the value of the i-th frequency from the frequency store.

int getN();

Gets the total number of elements that have been put into the store (the sum of the frequencies).

int operator ==(FrequencyStore<T> t);

Compares the contents of two frequency stores for equality.

int operator !=(FrequencyStore<T> t);

Compares the contents of two frequency stores for inequality.

int getSize();

Returns the number of distinct items in the frequency store.

Friend Implementor Functions and Operators

friend ostream & operator << (ostream &, FrequencyStore<T>&);

Produces a displayable representation of the frequency distribution.

Related Class: **FrequencyStoreIterator**

class FrequencyStoreIterator

template <class TY>
class FrequencyStoreIterator

Purpose: To **i**terate on a frequency store.

Manager Functions and Operators

FrequencyStoreIterator(FrequencyStore<TY> & s);

Implementor Functions

int atEnd();

TRUE if iterator is incremented beyond the end of the store, FALSE otherwise.

void operator ++ ();

Advances the iterator to the next item in the store. If the current item is the last one in the score, calling this function will make the current position of the iterator invalid. The **atEnd()** function can be used to test for this condition.

TY getValue();

Gets the current value from the frequency store.

int getFrequency();

Gets the current frequency from the frequency store.

Related Class: FrequencyStore

class Group

Inherits from public Rest

Purpose: To put a groupette marker in a score. A groupette consists of notes that are inserted in the normal way, but are preceded by a groupette marker which specified the number of units in the groupette, the type of units present and the length of the groupette in terms of its real duration.

Manager Functions and Operators

Group(void);

Group(int gn, Duration dr, durType gbu);

where **gn** is the number of basic units in the group, **dr** is the overall duration of the groupette, and **gbu** is the basic unit of the groupette. The notated duration of the groupette is **gn** x **gbu.** The actual duration is **dr**.

Group(const Group & gr);

Group & operator = (const Group & gr);

virtual ~Group();

Access Functions

int getGroupetteNumber(void) const;

Returns the number of basic units in the groupette.

void putGroupetteNumber(int n);

Sets the number of basic units in the groupette to **n**.

durType getGroupetteBasicUnit(void) const;

Returns the groupette basic unit.

void putGroupetteBasicUnit(durType dr);

Sets the groupette basic unit.

String getString();

Returns a string description of the groupette marker.

String getName();

Returns the string "Group".

class Instrument

Purpose: To associate an instrument with a stave.

Manager Functions and Operators

Instrument();

Instrument(const String & s, int tr = 0);

Creates an instrument object for instrument **s** which transposes up **tr** semitones.

int operator==(const Instrument & in);

virtual ~Instrument();

Access Functions

String getString();

Returns the name of the instrument.

String getName();

Returns the text "Instrument".

int getTranspose();

Returns the number of semitones by which the instrument is transposed up.

If a negative number is returned, the instrument is transposed down.

void putInstrument(const String & s);

Sets the name associated with the instrument object to **s**.

class KeySig

Purpose: To represent key signatures in a score

Manager Functions and Operators

KeySig(keySigType ks = NOKEY) ;
KeySig(const KeySig & ks);
virtual ~KeySig();

where keySigType is
enum
keySigType
{
C, SF, SFSC, SFSCSG, SFSCSGSD, SFSCSGSDSA, SFSCSGSDSASE,
SFSCSGSDSASESB, FB, FBFE, FBFEFA, FBFEFAFD, FBFEFAFDFG,
FBFEFAFDFGFC, FBFEFAFDFGFCFF, NOKEY
};

Access Functions

keySigType getKeySig(void) const;
Returns the key signature value.
void putKeySig(keySigType ks);
The key signature is set to ks.
String getString();
returns one of
"#F#C#G#D#A#E#B", "#F#C#G#D#A#E", "#F#C#G#D#A",
"#F#C#G#D","#F#C#G", "#F#C", "#F",
"YBYEYAYDYGYCYF", "YBYEYAYDYGYC", "YBYEYAYDYG",
"YBYEYAYD", "YBYEYA", "YBYE", "YB", ""
These are the textual equivalents of the **keySigType** identifiers listed above.
String getName();
Returns the string "Keysig".

Associated Function:

String ksToText(keySigType k_s);

Converts a key signature to a string.

Comment

Key signature has an open scoping mechanism.

union FAR MIDIMsg

Purpose: To represent MIDI messages.

Manager Functions and Operators

MIDIMsg(unsigned char m1=0, unsigned char m2 = 0, unsigned char m3 = 0);
Creates a one to three byte MIDI message.

Access Functions

void update(unsigned char m1, unsigned char m2, unsigned char m3);
Modifies the three data bytes in a MIDI message.
void update1(unsigned char m1);
Modifies the first byte of a MIDI message.
void update2(unsigned char m2);
Modifies the second byte of a MIDI message.
void update3(unsigned char m3);
Modifies the third byte of a MIDI message.
BYTE & getByte(int i);
Returns byte **i** of a MIDI message.
DWORD & getWord ();
Returns the MIDI message as a double word.
void putWord(DWORD w);
Writes the double word to the MIDI message.

The MIDI bytes are left alligned in the DWORD type. The righmost byte of the DWORD is not used.

class MIDIStream

Purpose: To implement a MIDI output stream.

MIDIoutStream & operator << (MIDIoutStream & mo, MIDIMsg mm);

Sends the MIDI message in **mm** to the MIDI output stream **mo**.

MIDIoutStream & operator << (MIDIoutStream & mo, unsigned int t);

Sends the three leftmost bytes of **t** to the MIDI output stream **mo**.

MIDIoutStream & operator << (MIDIoutStream & mo, int t)

Sends the three leftmost bytes of **t** to the MIDI output stream **mo**.

class Note Inherits from Rest and Pitch.

Purpose: To represent a note object in a score.

Manager Functions and Operators

Note(char pa = 'C', int oc = 5, accidType ac = NOACCID,
durType d = N8, int dot = 0, Set nr = Set());
pa = one of 'A', 'B', 'C', 'D', 'E', 'F', 'G'.
oc = octave register, with middle C = 5.

accidType is
enum
accidType
{
NOACCID, F, S, N, DF, DS
};

durType is
enum
durType
{
N0, N1, N2, N4, N8, N16, N32, N64, N128
};

dot = number of dots.

nr is a set which contains combinations of

STACCATO, TIE_FROM, TIE_TO, TENUTO, PLUS, FERMATA, COMMA,
PAUSE_MARK, TREMOLO, TREMOLO_END, GLISSANDO,
GLISSANDO_END, SQUARE_NOTEHEAD, DIAMOND_NOTEHEAD,
X_NOTEHEAD, OMIT_NOTEHEAD, OCTAVE_UP, OCTAVE_DOWN,
OCTAVE_END, ARPA, PIZZ, HARMONIC, COL_LEGNO, PONTICELLO,
PED, REL, OCTAVE_DOUBLE_UP, OCTAVE_DOUBLE_DOWN,

OCTAVE_DOUBLE_END, TURN0, TURN1, TURN2, TURN3, TURN4, TURN5, TURN6, TURN7, TURN8, TURN9, TURN, SLUR1, SLUR1_UP, SLUR1_DOWN, SLUR1_END, SLUR2, SLUR2_UP, SLUR2_DOWN, SLUR2_END, ACCENT, HEAVY_ACCENT, UP_BOW, DOWN_BOW, LETTER_TR, BAROQUE_TRILL, GRACE_NOTE, BEAM, UP_BEAM, DOWN_BEAM, BEAM_END, REST_ALLIGNMENT, ALTERNATE, PPPP, PPP, PP, PIANO, MF, FORTE, FF, FFF, FFFF, CRESCENDO, CRESCENDO_END, DIMINUENDO, DIMINUENDO_END.

Note(const Note & nt);
virtual ~Note() { }
Note & operator = (const Note & nt);

Access Functions

char getAlpha(void) const;
Returns the alphabetic note class name in the range 'A' to 'G'.
void putAlpha(char a);
Sets the alphabetic note class name to **a** where **a** is in the range 'A' to 'G'.
int getOctave(void) const;
Returns the octave register number. The register starting at middle C is 5.
void putOctave(int o);
Sets the octave register number to **o**.
accidType getAccid(void) const;
Returns the accidental value associated with the note.
void putAccid(accidType a);
Sets the accidental value of the note to **a.**
void clearAttributeSet(void);
Clears all attributes of the note.
void putAttribute(const nrAttrType & nr);
Sets the attribute **nr** in the attributes set of the note.
void putAttributeSet(const Set & s);
Sets the attribute set of the note to the set **s**.
Set getAttributeSet(void) const;
Returns the attribute set for the note.
durType getHead(void) const;

Returns the **durType** value of the note.

void putHead(durType d, int dts = 0);

Sets the time value of the note to a **durType** value of **d** and the number of dots to **dts**.

int getDots(void) const;

Returns the number of dots of the note.

void putDots(int d);

Sets the number of dots of the note.

Rat getRDuration(void) const;

Returns the duration as a rational number, without any groupette scoping taken into account.

const Duration & getDuration(void) const;

Returns the duration of the note head.

int getPitch12(void) const;

Returns the chromatic pitch number of the current note. Middle C corresponds to 60. No scoping information is taken into account.

int getPitch7(void) const;

Returns the diatonic pitch number. Middle C corresponds to 35.

String getString();

Returns a string description of a Note object in the form

<Pitch part description><Duration part description><Attributes>.

String getName();

Returns the string "Note".

Comment

The exact placement of a dynamic on a note is not specified. In some cases, for example for creschendi and diminuendi, this may prove inadequate. A further study is called for here.

class PartsExpert

Purpose: To identify the parts in a Irish dance tune.

Manager Functions and Operators

PartsExpert(Score & s);
Creates a parts expert object for score s.

Implementor Functions

int isSingled();
Returns TRUE if the tune is singled, FALSE otherwise.

int numberOfParts();
Returns the number of parts in the tune.

int hasOddPart();
Returns TRUE if the tune has an uneven number of parts, FALSE otherwise.

int getBarNoForPart(int i);
Returns the bar number at which part i starts. Prints an error message to **cerr** if it is called for an invalid value of **i.**

Comment

See chapter 7.6 for details of this class.

class Pitch

Purpose: To represent a pitch abstraction. It differs from class **Note** in that it does not have either a duration or a set of attributes, and also does not exist with a scoping context.

Manager Functions and Operators

Pitch(char pa = 'C', int oc = 5, accidType ac = NOACCID);

accidType is
enum
accidType
{
NOACCID, F, S, N, DF, DS
};

Pitch(const Pitch &);
virtual ~Pitch();

Access Functions:

char getAlpha(void) const;
Returns the alphabetic note class name of the pitch in the range 'A' to 'G'.
void putAlpha(char a);
Sets the alphabetic note class name of the pitch to **a** where **a** is in the range 'A' to 'G'.
int getOctave(void) const;
Returns the octave register number. The register starting at middle C is 5.
void putOctave(int o);
Sets the octave register number to **o**.
accidType getAccid(void) const;
Returns the accidental value associated with the pitch.
void putAccid(accidType a);
Sets the accidental value of the pitch to **a.**

String getString();

Returns a string description of the pitch in the form of <alphabetic Letter> <octave Number> <accidental name>.

String getName();

Returns the string "Pitch".

Implementor Functions

int getPitch12(void) const;

Returns the chromatic pitch number of the current note. Middle C corresponds to 60.

int getPitch7(void) const;

Returns the diatonic pitch number. Middle C corresponds to 35.

The next set of overloaded operators perform magnitude comparisons of pitch.

int operator > (const Pitch & pt);

int operator < (const Pitch & pt);

int operator == (const Pitch & pt);

Related Classes: Note

class PitchClasses

Purpose: To represent pitch class sets.

Manager Functions and Operators

```
PitchClasses ( int a0 =-1,int a1 =-1,int a2 =-1,int a3 =-1,int a4 =-1,
    int a5 =-1,int a6 =-1,int a7 =-1,int a8 =-1,int a9 =-1,
    int a10=-1,int a11=-1,int a12=-1,int a13=-1,int a14=-1,
    int a15=-1,int a16=-1,int a17=-1,int a18=-1,int a19=-1,
    int a20=-1,int a21=-1,int a22=-1,int a23=-1,int a24=-1,
    int a25=-1,int a26=-1,int a27=-1,int a28=-1,int a29=-1,
    int a30=-1,int a31=-1);
PitchClasses( const  PitchClasses &);
PitchClasses(const Set & );
PitchClasses & operator = (const PitchClasses & );
```

Example

```
PitchClasses x(0, 2, 4, 5, 7, 9, 11);
```

Creates a pitch class set of the notes of the diatonic scale.

Access Functions

```
void init(int i = 0);
```

This function does nothing. It exists for compatibility with class **FrequencyStore**.

Implementor Functions

```
void pitchClass(Score &);
```

Adds the pitch classes found in the score to the set.

void pitchClass(ScoreIterator &si1, ScoreIterator &si2);

Add the pitch classes to the set that are found in the score of **si1**, starting with **si1**, and continuing until either **si2** is reached, or else until the end of of the score is reached. **si2** may be **NULL**.

void pitchClassInc(ScoreIterator & si);

If **si** points to a note, the pitch of that note is added to the pitch class set object, otherwise no action is taken.

int bestNormalElement();

The pitch of the first normal element calculated and returned.[140]

void primeForm();

The set is converted to its prime form.

void nIEPrimeForm();

The set is converted to its non inversionally equivallent prime form.[141]

void invert();

Converts the pitch class set to its inversion.

String getPFName();

Returns the name of the pitch class set.[142]

String getNIEPFName();

Returns the name of the non inversionally equivallent pitch class set.[143]

Friend Implementor Functions and Operators

friend ostream& operator << (ostream& , PitchClasses);

Outputs a representation of the pitch class set in a form suitable for display.

---

[140] See Alan Forte The Structure of Atonal Music (New Haven: Yale UP 1973, 1979 printing), p4.

[141] See 8.1.

[142] Alan Forte, op.cit., pp179-181.

[143] See 8.1.

class PitchTuple Inherits from class Tuple

Purpose: To represent a set of pitches, relative to the initial one, which is set at 0.

Manager Functions and Operators

PitchTuple();
PitchTuple(int sz);
PitchTuple(const PitchTuple & pt);

Access Functions

void put(int t, int i);
Puts the value **t** into position **i** of the tuple. The first position in the tuple corresponds to **i** = 0, and this position must be filled first.

Friend Implementor Functions and Operators:

ostream & operator << (ostream & os, const PitchTuple &t);
Outputs the tuple in a form suitable for display.

Related class: Tuple

class Q

const int Q_TEMPLATE_SIZE = 10;

template < class QT >
class Q

Purpose: To implement an array based implementation of a first in first out store.

Prerequisites: The type of QT must have a valid assignment, ==, and > operator, and a default constructor.

Manager Functions and Operators:

Q(int sz = Q_TEMPLATE_SIZE);
Creates an array-based queue with initial size **SZ.**
~Q();
Q(const Q & q);
Q & operator = (const Q & qIn);

Access Functions

int isEmpty();
Returns **TRUE** if queue is empty, **FALSE** otherwise.
void put(QT c);
Puts a copy of **c** into the queue.
QT get();
Returns a copy of the item at the head of the queue, and deletes the item from the queue.
qSize();
Returns the number of items in the queue.

class Rat

Purpose: To represent rational numbers.

Manager Functions and Operators:

Rat(long n1 = 0 , long d1 = 1);
Rat(const Rat & r);
virtual Rat & rationalize();
const Rat & operator = (const Rat & r);

operator int();
operator long();
operator float();

Access Functions

const Rat & rmin(const Rat & r1, const Rat & r2);
Returns a reference to the minimum of r1 and r2.
const Rat & rmax(const Rat & r1, const Rat & r2);
Returns a reference to the maximum of r1 and r2.
long numer() const;
Returns the value of the numerator.
long denom() const;
Returns the value of the denominator.
void putNumer(int num);
Changes the numerator to **num.**
void putDenom(int den);
Changes the denominator to **den.**

Implementor function and operators

Member arithmetic operators.
const Rat & operator += (const Rat & b);
const Rat & operator -= (const Rat & b);

const Rat & operator *= (const Rat & b);
const Rat & operator /= (const Rat & b);

Friend Implementor function and operators

Friend arithmetic operators.
friend Rat operator - (const Rat & a);
friend Rat operator + (const Rat & a);
friend Rat operator + (const Rat & a, const Rat & b);
friend Rat operator - (const Rat & a, const Rat & b);
friend Rat operator * (const Rat & a, const Rat & b);
friend Rat operator / (const Rat & a, const Rat & b);
friend int operator > (const Rat & r1, const Rat & r2);
friend int operator >= (const Rat & r1, const Rat & r2);
friend int operator < (const Rat & r1, const Rat & r2);
friend int operator <= (const Rat & r1, const Rat & r2);
friend int operator == (const Rat & r1, const Rat & r2);
friend int operator != (const Rat & r1, const Rat & r2);

Stream friend functions.
friend ostream& operator << (ostream& os, const Rat & r);
Outputs the object in a form suitable for display.
friend istream& operator >> (istream& is, Rat& r);
Inputs a rational number expressed in the form

{ <numerator> , <denominator }.

Related Classes: TimeSigType.

class Rest **Inherits from Duration**

Purpose: To represent a rest object in a score.

Manager Functions and Operators

Rest(durType d = N8, int dot = 0, Set e = Set());

where **durType** is
enum durType { N0, N1, N2, N4, N8, N16, N32, N64, N128 }

dot is the number of dots on the rest

The set **e** is made up of members with appropriate combinations of the nrAttrTypes, possibly **FERMATA**, **BREATH_MARK** and **ALTERNATE.** The last value is used only with crotchet rests to indicate the 'reversed 7' English notation.

Rest(const Rest & r);
virtual ~Rest();
Rest & operator = (const Rest & rst);

Access Functions

void clearAttributeSet(void);
Clears all attributes of the rest.
void putAttribute(const nrAttrType & nr);
Sets the attribute ***nr*** in the attributes set of the rest.
void putAttributeSet(const Set & s);
Sets the attribute set of the rest to **s**.
Set getAttributeSet(void) const;
Returns the attribute set for the rest.
durType getHead(void) const;
Returns the **durType** value of the rest.
void putHead(durType d, int dts = 0);

Sets the time value of the rest to a **durType** value of **d** and the number of dots to **dts**.

int getDots(void) const;

Returns the number of dots of the rest.

void putDots(int d);

Sets the number of dots of the rest.

Rat getRDuration(void) const;

Returns the duration as a rational number, without any groupette scoping taken into account.

const Duration & getDuration(void) const;

Returns the duration of the rest.

String getString();

Returns a string description of the Rest in the form of

<Duration part description><Attributes>.

String getName();

Returns the string "Rest".

class Score

Purpose: To represent a score.

Manager Functions and Operators

Score(void);
Creates a null score.
Score(String filename);
Creates a score from the contents of filename.
Score(const Score &);
~Score(void);
Score& operator = (const Score & s);
void operator - (void);
The unary operator - is used to make a score null, that is it deletes all the objects in the score, but it does not destroy the score object itself.

Access Functions

void setMaxStaves(const int & n);
Initialises the number of staves in a score.
int getNoStavesInScore(void);
Returns the number of staves in a score.
int getInitialBarNo() const;
Returns 0 if there is an incomplete 1st bar, and 1 otherwise. An initial anacrusis is regarded as being in bar 0.
Rat getInitialPosition() const;
Returns the location of the first note or rest in the score. For example a score in 6/8 time with an initial quaver as an anacrusis, this will return 5/8.
String getString(const tagType & tt = TAG) const;
String getString(const String & tt) const;
Both of these functions return a string value of the attribute.

Where **tt** is one of **COMPOSER, PUBLISHER, EDITOR, MANUSCRIPT, COLLECTOR, PERFORMER, COMMENTS, CATEGORY, TITLE, ETITL,**

NUMBER, MOVEMENT, WORDS, METRONOME, TEMPO, EXPRESSION, INSTRUMENT.

Note: **TITLE** and **ETITL** allow for a main title and a subsidiary title. The second version of the getString functions is provided to allow for future expansion, so that new fields may be created, over and above those provided by the system.

void putString(const String & text, tagType tt)

Puts the tagged text into the score object. Valid tags are as for **tt** above.

clefType getClef(void) const;

Returns the first clef on the first stave of the score.

keySigType getKeySig(void) const;

Returns the first key signature on the first stave of the score.

long getTimeSigNumerator(void) const;

Returns the numerator of the first time signature on the first stave of the score.

long getTimeSigDenominator(void) const;

Returns the denominator of the first time signature of the first stave of the score.

TimeSigType getTimeSig(void) const;

Returns the time signature value of the first time signature of the first stave of the score.

int isNull(void) const;

Returns TRUE if the score is a null score This may arise when an attempt is made to get a score from a non-existent file, or when the **Score(void)** constructor is used.

class ScoreIterator

Purpose: To an iterator on a score.

Manager Functions and Operators

ScoreIterator(const Score & s, int lid = -99);

Creates a score iterator for the score **s.** The type of score iterator and its associated mode depends on the type of score that it is created for.

If the second parameter is absent and the score consists of a single stave, then a single stave iterator in **MONO** mode is created. That is the iterator follows the uppermost pitches on the stave.

If the second parameter is absent and the score consists of multiple staves, then a multi-stave iterator in **POLY** mode is created.

If the second parameter is present it must be a stave number in the range 0 to the (maximum number of staves - 1). In this case a single stave iterator in **MONO** mode is created for the specified stave. See 5.10 for more details of the operation of these operators.

ScoreIterator(const ScoreIterator & si);

ScoreIterator(int lid = 0);

ScoreIterator::~ScoreIterator(void);

const Score & score();

ScoreIterator& operator = (const ScoreIterator & si);

Access Functions

Information extracting functions.

tagType getTag();

Returns the tag value retrieved, which is one of

NOTE, REST, GROUP, WORDS, BARLINE, CLEF, TIME_SIG, KEY_SIG, TEXT, METRONOME, TEMPO, EXPRESSION, START, INSTRUMENT

int isA(tagType t);

Returns TRUE if the current object has type **t.**

clefType getClef(void) const;

Returns the current clef.

keySigType getKeySig(void) const;

Returns the current key signature.

long getTimeSigNumerator(void) const;

Returns the numerator of the current time signature.

long getTimeSigDenominator(void) const;

Returns the denominator of the current time signature.

TimeSigType getTimeSig(void) const;

Returns the current time signature.

String getString(const tagType & tt = TAG) const;

gets a string description of the object or scope indicated by **tt**, where is one of

COMPOSER, PUBLISHER, EDITOR, MANUSCRIPT, COLLECTOR, PERFORMER, COMMENTS, CATEGORY, TITLE, ETITL, NUMBER, MOVEMENT,TAG, NOTE, REST, WORDS, BARLINE, CLEF, TIME_SIG, KEY_SIG, TEXT, METRONOME, TEMPO, EXPRESSION, INSTRUMENT

String getString(const String & tagString) const;

Returns a string representation of the current object.

String getName() const;

Returns the name of the current object. It is one of

void putString(const String & text, tagType tt);

Puts the tagged string text into the tagged field **tt**, where **tt** is one of

COMPOSER, PUBLISHER, EDITOR, MANUSCRIPT, COLLECTOR, PERFORMER, COMMENTS, CATEGORY, TITLE, ETITL, NUMBER, MOVEMENT, WORDS, METRONOME, TEMPO, EXPRESSION, INSTRUMENT.

Functions for retrieving values in notes and rests.

char getAlpha(void) const;

Returns the alphabetic note class name in the range 'A' to 'G'.

void putAlpha(char a);

Sets the alphabetic note class name to **a** where **a** is in the range 'A' to 'G'.

int getOctave(void) const;

Returns the octave register number. The register starting at middle C is 5.

void putOctave(int o);

Sets the octave register number to **o**.

accidType getAccid(void) const;

Returns the accidental value immediately associated with the note.

int getKeySigAdjust() const;

Returns the number of simitones by which the current key signature displaces the pitch of the note.

accidType getAccidAlterationInBar() const;

Returns any local accidental that influences the pitch of the note.

void putAccid(accidType a);

Sets the accidental value of the note to **a.**

void clearAttributeSet(void);

Clears all attributes of the note or rest.

void putAttribute(const nrAttrType & nr);

Sets the attribute **nr** in the attributes set of the note or rest.

void putAttributeSet(const Set & s);

Sets the attribute set of the note or rest to **s**.

Set getAttributeSet(void) const;

Returns the attribute set for the note or rest.

durType getHead(void) const;

Returns the **durType** value of the note or rest.

void putHead(durType d, int dts = 0);

Sets the time value of the note or rest to a **durType** value of **d** and the number of dots to dts.

int getDots(void) const;

Returns the number of dots of the note or rest.

void putDots(int d);

Sets the number of dots of the note or rest.

Rat getRDuration(void) const;

Returns the duration as a rational number, with resolution of groupette scoping.

const Duration & getDuration(void) const;

Returns the duration of the note head without resolution of groupette scoping.

int hasAttribute(const nrAttrType & na) const;

Returns TRUE if the attribute **na** is present in the current entitiy.

ScoreIterator getNext();

Returns a score iterator which points to the next entity in traversal order in the score.

Rat getRemainder(void) const;

Returns the time distance between the current position and the end of the entitiy pointed to.

int getPitch12(void) const;

Returns the chromatic pitch number of the current note, with all scoping taken into account. Middle C corresponds to 60.

int getPitch7(void) const;

Returns the diatonic pitch number. Middle C corresponds to 35.

Pitch getPitch() const;

Returns the pitch entitiy associated with the current note.

String getWords(void) const;

Retrieves the vocal text.

long getBarNo(void) const;

Returns the current bar number.

barType getBarType(void) const;

If the current entity is a bar, the bar type is returned.

nrAttrType getDynamics(void);

Returns the dynamic value associated with the current note.

Rat getBarDist(void) const;

Returns the rational distance of the **scoreIterator** from the start of the current bar.

Information relating to groupettes.

Rat getGroupetteRemaining(void);

Returns the rational distance from the **scoreIterator** to the end of the current groupette.

long getGroupetteNumber(int depth = 0) const;

Returns the number of units in the groupette.

durType getGroupetteBasicUnit(void) const;

Return the notated time value of the basic units in a groupette.

Testing functions and operators returning TRUE(=1) or FALSE (=0).

int isFirst(void);

Returns TRUE if current entity is the first on its stave.

int isLast(void);

Returns TRUE if current entity is the last on its stave.

int isNullStave();

Returns **TRUE** if the current stave is empty, **FALSE** otherwise.

Rat getTimeSlice(void);

Returns the rational distance between the current position and the nearest entity in time.

scanModeType getScanMode();

Returns the scan mode of the iterator.

void putScanMode(scanModeType sm);

Set the scan mode for the iterator to **sm.**

Implementor Functions and Operators

Navigation functions.

int step(tagType gt = ANY);

Moves the current position to the next entity in traversal order. Returns **TRUE** if successful. See 5.10 for details of traversals. **gt** may have one of the following values

NOTE, REST, WORDS, BARLINE, CLEF, TIME_SIG,
KEY_SIG, TEXT, TEXT_OBJECT, START, ANY

int mstepb(tagType gt = ANY);

Operates in a manner similar to the last function but moves the iterator in the opposite direction. Returns **TRUE** if successful.

int step(Rat d);

Moves the iterator to the first score object that is at a distance d from the current position. Returns **TRUE** if successful. Note that the ends of notes or rests are not candidate objects for moving the iterator to.

int locate(const tagType & gt = ANY, const int & n = 1);

Moves the iterator to the specified location in the score. Returns **TRUE** if successful. **gt** may be one of

NOTE, REST, WORDS, BARLINE, BAR, CLEF, TIME_SIG, KEY_SIG, TEXT, METRONOME, TEMPO, EXPRESSION, START, ANY

BARLINE is used to specify a count of actual barlines. **BAR** is used to specify a specific bar by number.

The locate function locates the **n**th entity of type **gt** in the score.

Hence

si.locate(NOTE, 20);

will position the score iterator at the start of the 20th note in the score.
locate() positions the iterator at the start of the score.

Score building and manipulation.

The binary '^' operator is used for inserting single entities into scores at the current position and the corresponding unary operator '!' is used to remove the entity at the current position. The '+' operator inserts a new object after the current position.

void operator + (Instrument & in);

Adds an instrument designation to a stave.

void operator + (Barline & nbl);

Adds a barline to the current position on the current stave.

void operator + (Text & t);

Adds a textual entry to the current position on the current stave.

void operator + (Note & n);

Adds a note object to the current position on the current stave.

void operator + (Rest & r);

Adds a rest object to the current position on the current stave.

void operator + (Group & g);

Adds a groupette marker to the current position on the current stave.

void operator + (TimeSig & ts);

Adds a time signature object to the current position on the current stave.

void operator + (Clef & cl);

Adds a clef object to the current position on the current stave.

void operator + (KeySig & ks);

Adds a key signature object to the current position on the current stave.

void operator ^ (Note & nt);

Attaches a note object vertically to the current note or rest object.

void operator ^ (Rest & rst);

Attaches a rest object vertically to the current note or rest object.

void operator ^ (Group & gr);

Attaches a groupette object vertically to the current groupette object.

void operator ^ (Words & wd);

Attaches words to the current note object.

void setPlayOn(void);

Causes automatic output of each traversed note to the current MIDI device.

void setPlayOff(void);

Switches off automatic MIDI output.

unsigned long getPlaySpeed(void) const;

Returns the curent playing speed setting.

void putBorrow(long l);

Causes the duration of the MIDI output for the next note to be reduced by **l** ticks. This is used typically to allow for the playing of grace notes while keeping the overall rhythm intact.

long getBorrow(void) const;

Returns the duration of the MIDI next note shortening. See putBorrow(long l).

void setPlayMetronome(int mm);

Sets the play speed to the indicated metronome value, given in terms of the number of quarter notes per minute. The default is 200.

Implementor Friend Functions and Operators

IO operators.

friend ostream& operator << (ostream& is, const ScoreIterator & si);
Outputs a textual representation of the current object.

Comparison operators to compare positions on the basis of location in absolute score time. It is the responsibility of the user to ensure that **si1** and **si2** are positions in the same score.

friend int operator > (const ScoreIterator & si1, const ScoreIterator & si2);
friend int operator >= (const ScoreIterator & si1, const ScoreIterator & si2);
friend int operator < (const ScoreIterator & si, const ScoreIterator & si2);
friend int operator <= (const ScoreIterator & si1, const ScoreIterator & si2);
friend int operator == (const ScoreIterator & si1, const ScoreIterator & si2);
friend int operator != (const ScoreIterator & si1, const ScoreIterator & si2);

void traverse(ScoreIterator & si1, ScoreIterator & si2, Rat &r);

The traverse function is used to move through two sections of scores, one time slice at a time. Each call to traverse advances the iterators **si1** and **si2** by the same amount, equal to the current time slice.

class Set

Purpose: To represent sets of integers or enumerated types.

Manager Functions and Operators:

Set (
int a0 =-1,int a1 =-1,int a2 =-1,int a3 =-1,int a4 =-1,
int a5 =-1,int a6 =-1,int a7 =-1,int a8 =-1,int a9 =-1,
int a10=-1,int a11=-1,int a12=-1,int a13=-1,int a14=-1,
int a15=-1,int a16=-1,int a17=-1,int a18=-1,int a19=-1,
int a20=-1,int a21=-1,int a22=-1,int a23=-1,int a24=-1,
int a25=-1,int a26=-1,int a27=-1,int a28=-1,int a29=-1,
int a30=-1,int a31=-1,int a32=-1,int a33=-1,int a34=-1);

Constructs a set with the parameters as members.
Example

Set s(2, 4, 6, 8); // creates a set of 4 integers
Set s(); // creates a null set.

enum Weekdays { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY};
Set workingDays();

Set (const Set &);
Set & operator = (const Set &);
operator String();

Access Functions

void setEnumLimit(int i);
This function set a guard on the allowable values in the set. i should be the maximum number of enumerated value allowable. It is advisable to initialise all sets with

this functions. The complement operator will not work properly unless this is done.

For example, in the above set, it is desirable to invoke the function

workingDays.setEnumLimit(FRIDAY);

Set s1 = Set (MODAY, TUSEDAY, WEDNESDAY, THURSDAY, FRIDAY);

Set workingDays = Set(SATURDAY); // Error

int getEnumLimit(void) const;
Returns the ordinal value of the highest constant allowable in the set.

int card() const;
Returns the cardinal number of a set.

Implementor Functions

String getString();
Returns a textual representation of the set.

int operator < (Set &s);
Prividesan unspecified ordering on the set.

Implementor Friend Functions and Operators

friend Set operator + (const Set & s1, const Set & s2);	union
friend Set operator * (const Set & s1, const Set & s2);	intersection
friend Set operator - (const Set & s1, const Set & s2);	difference
friend int operator < (int i1, const Set & s1);	membership
friend int operator <= (const Set & s1, const Set & s2);	subset
friend int operator == (const Set & s1, const Set & s2);	equality
friend int operator != (const Set & s1, const Set & s2);	inequality
friend Set operator ~ (const Set & s1);	complement

Stream function overloads.

friend ostream& operator << (ostream& os, Set s);
friend istream& operator >> (istream& is, Set& s);

Related Classes: PitClass and SetIterator

class SetIterator

Purpose: To iterate on a set.

Manager Functions and Operators

SetIterator(Set & s);
Creates an iterator for set s.

Access Functions

const int & getCurrent();
Gets the ordinal value of the current element.

void makeCurrent(const int & c);
Makes element with ordinal value **c** the current one.

int next();
Moves the iterator cyclically to the 'next' element in the set.

void init();
Sets the iterator to the 'first' element of the set.

int atEnd();
True if the iterator is at the 'last' element of the set.

class Stack

template <class T>
class Stack

Purpose: To implement a first in last out store.

Manager Functions and Operators

Stack(const int s = 100);

Creates a stack using an array of size **s.**

Stack(const Stack & s);

Stack & operator = (const Stack & s);

~Stack();

int inStack(const T & p) const;

void flush();

Access Functions

void push(T p);

Pushes a copy of **p** onto the stack. The array will double in size on overflowing. If there is not enough space for expanding the array, the function sends an error message to **cerr** and terminates the program.

T pop();

Returns a copy of the entity on the top of the stack, and removes it from the stack.

int isFull(void);

Returns **FALSE** always.

int isEmpty(void);

Returns **TRUE** if the stack is empty, **FALSE** otherwise.

class String

Purpose: To represent character strings.

Manager Functions and Operators:

String(int len = MAXCSLEN)

Constructs a null string of length specified by **len.**

String (char * st)

Constructs an instance of class string from a c string.

Examples

```
String s1;
String s2(20);
String s3("demo");
```

String(const String & s)

Creates a deep copy of string s.

~String()

String operator = (const String & s);

String operator = (char c);

String & cvtNs(int i);

Converts the integer to its string representation.

int num();

Converts a string to its integer representation.

String & operator = (const StringIterator & sti);

Access Functions

char & operator[] (int i)

Indexing of individual characters in a string.

Implementor Functions

int length() const;

Returns the number of characters in a string.

char * getCString( ) const;

Returns a c-string. Care must be exercised in the use of the returned pointer which should be limited to within the lifetime of the object.

String operator + (char c)

Returns a concatenation of the object and character **c**.

String operator + (const String & s)

Returns a concatenation of the object and string s.

String & operator += (const String & s);

Concatenates the object and **s**.

String & operator += (char c);

Concatenates the object and **c**.

String & pad(int n, char c)

Makes the string to consist of **n** characters of value **c**.

The following comparison operators are available which yield **TRUE/FALSE** values, where the ordering is done on the basis of ASCII colating sequences.

int operator == (const String & st)
int operator != (const String & st)
int operator > (const String & st)
int operator < (const String & st)
int operator >= (const String & st)
int operator <= (const String & st)

int index(char c) const

Returns the index of the first character in the string equal to c. Returns a negative number if the character is not found.

int index(String & s)

Searches the object for the first occurrence of the string **s** and returns the index of the start of the string in the object. Returns a negative number if the character is not found.

void sub (int i1, int i2)

Converts the object int a new string consisting of its substring from index **i1** to index **i2**.

String & upperCase()

Converts the object to upper case.

Friend Operators

ostream & operator << (ostream & os, const String & st);

Stream output.

istream & operator >> (istream & is, String & st);

Stream input.

Related Classes: StringIterator

Instances

String NULLSTRING

class StringIterator

Purpose: To traverse class String

Manager Functions and Operators and Operators

StringIterator(String & s);
Creates an object of class StringIterator for a string s, with it current position set at the first character in the string.

StringIterator();
Creates an unassigned object of class StringIterator.

Copy Constructors - default used.

Destructors - default used.

Assignment - default used.

Access Functions

char * getCString() const;
Gets the C-string part of object. Be careful here with lifetimes here.

char get() const;
Gets character at current position.

void put(char c);
Replaces character at current position. If current position is at null, then it appends the character to the string.

Implementor Functions and Operators

int atEnd() const;
Tests for the end of string condition happens, **TRUE** is returned if the Iterator is at the final null.

int operator ==(const StringIterator & s);
Tests the equality of two substrings, from current position to end of string.

int operator != (const StringIterator & s);
Tests for inequality of two substrings, starting at their current position.

StringIterator & operator ++();

Increments the current position Its maximum value corresponds to the final null.

StringIterator & operator --();

Decrements the current position, if the current position is to the right of the first.

StringIterator & operator += (int i);

Compound of + and =.

void reset();

Sets the cursor to start of string.

StringIterator & next();

Advances iterator to first non-blank character in the string.

int searchFor(char c);

Returns **TRUE** if **c** in substring and relocates iterator to position of **c.**

int isKeyWord(int & count, String const ar[], const int & len);

Returns **TRUE** if object is one of the characters in **ar**. **len** is the length of the array, and **count** is the matching index of the array.

int nextChar(const char & c);

If next non blank character in String is **c**, it increments **p** and returns **TRUE.**

int heads(String tstr);

Returns **TRUE** if iterator points to part of a string that starts with **tstr.**

int betweenBrackets(String & s);

BetweenBrackets returns **TRUE** if the string has a left bracket '(' as its first non blank character and has matching right bracket and **FALSE** otherwise. If **TRUE**, the String **s** is a copy of the string that was enclosed between brackets.

int isNum();

Returns **TRUE** if the next non-blank entity in the string, starting at the current position is an integer.

int num();

Interprets the characters as a number and returns the value found. Advances the current position to the position following the number.

void outString(StringIterator si2 = NullString);

Writes out the substring, starting at the current position, until the end of the string is encountered or until position si2 is reached.

String makeString(StringIterator se = NullString);

Returns an object of type String consisting of the substring in the iterated object from the current position until position **se** is reached or to end of the string.

void upperCase();

Converts characters in the original string to uppercase, starting with the current position and ending with the end of string.

int length();

Returns the length of the original string, starting with the current position.

int uninstantiated();

Returns **TRUE** if the StringIterator is not currently associated with a string.

Friends

ostream & operator << (ostream & os, const StringIterator & si);

Outputs the original string, starting at the current position.

class Text

Purpose: To represent text objects in a score. Text objects are those which are not handled by class Word and do not use open scoping. Text entries in a score other than tempo, expression and metronome markings are represented by **class Text**.

Manager Functions and Operators

Text(tagType tg);
Text(const Text & tx);
virtual ~Text();

Text & operator = (const Text & tx);

Access Functions

The following function set and retrieve the information in a Text object.

String getText(void) const;
tagType getTextTag(void) const;
void putText(const String & s);
void putTextTag(tagType tt);
String getString();
Is the same as getText.
String getName();
Returns the string "Text".

class TimeSig

Purpose: To represent time signature in a score.

Manager Functions and Operators

TimeSig(int n1 = 4, int d1 = 4);
Creates a time signature of n1/d1.

TimeSig('C');
Creates a common time signature.

TimeSig('c');
Creates a common time signature.

TimeSig(const TimeSig & ts);

virtual ~TimeSig();

operator Rat() const;

Access Functions

long getTimeSigNumerator(void) const;
Returns the time signature numerator.

long getTimeSigDenominator(void) const;
Returns the time signature denominator.

TimeSigType getTimeSig();
Returns a copy of the time signature.

void putTimeSig(long n1, long d1);
Sets the time signature to **n1/d1.**

String getString();
Returns a string description of the key signature. It is of the form numerator, denominator, for example {3,4}.

String getName();
Returns the string "TimeSig".

Related Classes: TimeSigType

class TimeSigType Inherits for class Rat

Purpose: To represent rational numbers as in a time signature. Unlike objects of class Rat, the class that represents rational numbers. **TimeSigType** rational numbers are not automatically made relatively prime. Hence 6/8 does not automatically become 3/4, as it would for class Rat.

Manager Functions and Operators

TimeSigType(long n1 = 4, long d1 = 4);
TimeSigType(Rat rt);
TimeSigType & operator = (const TimeSigType & tst);

Access Functions

long getTimeSigNumerator(void) const;
Returns the numerator of the time signature.
long getTimeSigDenominator(void) const;
Returns the denominator of the time signature.
String getString();
Returns a string containing two numbers separated by a comma, for example, {6,8}.
String getName();
Returns the string "TimeSigType".

Related Classes: Rat, TimeSig.

class Tuple

Purpose: To represent Tuples, that is an ordered collection of integers, where the position in the tuple is deterimined at input.

Manager Functions and Operators

Tuple();
Used for creating null tuples.

Tuple(int sz);
Used for creating a sz-tuple.

Tuple(const Tuple & t);

~Tuple();

void init(const int tupleSize);
Used to populate a null tuple with tuples of size **tupleSize.**

Tuple & operator = (const Tuple & t);

operator String();

Access Functions

int operator [](int i);
Returns the value of the element at position i.

void put(int t, int i);
Writes t in the element at position i.

int operator == (const Tuple t);
Test for equality.

int operator != (const Tuple t);
Test for inequality.

int operator < (const Tuple t);
Provides an undefined ordering of tuples.

Friend Implementor Functions and Operators

ostream & operator << (ostream & os, const Tuple & t);
Stream output of a tuple in a displayable form.

Related Classes: PitchTuple

class Words

Purpose: To represent sung text in a score.

Manager Functions and Operators

Words(const String & t = String());
Words(const Words & wd);
virtual ~Words() { }

Words & operator = (const Words & wd);

Access Functions

The following function set and retrieve the field in a Word object.

String getWords(void) const;
Returns the current value.

void putWords(const String & s);
Sets the object to value **s**.

String getString();
Is the same as **getWords()**.

String getName();
Returns the string "Word".

Manual Pages for Functions.

float diff1 (ScoreIterator &, ScoreIterator &, Rat ln);
float difference (ScoreIterator &, ScoreIterator &, Rat ln);

These functions calculate the arithmetic difference between sections of length, **ln,** of two scores pointed to be **si1** and **si2**. See 8.3 for details of how the calculations are performed.

void setTrace();

Causes the of difference values calculated to be output to file 'out.out'.

void setDuration();

Causes each windowed difference to be weighted according to the duration of the note(s) that start at the beginning of the window.

Void setWindow();

Causes each windowed difference to be weighted by the width of the window.

void setSlopes();

Causes contour slopes to be incorporated into calculations.

void setStresses();

Causes metrical stresses to be incorporated into calculations.

void setTranspose();

Causes transposition independent calculations to be made.

Each function above has corresponding unset function, and query functions. The style of all of these follows the same pattern, as follows -

void unsetSlope();
int isSlopeSet();

diff1 is a simplified version of the **difference** function which estimates a melodic difference on the basis of pitch differences weighted by window widths only. Transposition processing is not done.

void form(String & str, Score & s, Rat In, float (*diff)(ScoreIterator &si1, ScoreIterator &si2, Rat ln), float criticalValue, int lid = 0);

Calculates the form of the monophonic score **s** using the function **diff** to calculate the differences. **ln** is the time interval of the segments used to evaluate the form. See 8.4 for details.

Various functions influence the effect of the difference calculations. See manual page for difference for details.

int getNextScoreNames(String fname, String & str, int depth = 0);

The purpose of this function is to examine the file **fname** which may contain either (a) an encoded score or (b) a list of files which contain encoded scores.

getNextScoreNames is called repeatedly until it returns **FALSE**. It places the name of the next file for processing in the string **str**, and returns **TRUE**. If no more files exist, the function returns **FALSE**. Hence this function may be useful in the following context

```
while ( getNextScoreNames( fname, str)
{
 Score s(str);

 . . . . do processing on s

}
```

In the case of **fname** containing a score, **getNextScoreNames** returns **TRUE** on the first call and copies fname to scr. In this case, a second call will return **FALSE**.

If **fname** contains a list of filenames, one per line, the first call to **getNextScoreNames** will return **TRUE**, and place the name of the first file in **str**. The subsequent calls will place the name of each next file in **str**. After the last file name form in **fname** is processed, the next call will return FALSE.

void isort(int inAr[], int outAr[], int size);

Performs an insertion sort on the first **size** elements of **inAr**, and produces the results in **outAr**.

int median(int values[], float weights[], int size);

Returns the median of the first **size** values in the array **values**, with the associate weights in array **weight**.

int sameSection(const Rat & ts, const Rat & p1, const Rat & p2);

Returns TRUE if **p1** and **p2** lie within the same primary division of **ts**. The primary division is determined by dividing **ts** by the lowest prime number of its denominator, other than one.

The function is of use for checking where two position within a bar fall in relation to the primary division of the bar.

Example:

Suppose the time signature is 6/8, then the primary division of the bar is by 2. In this case the following results are produced.

p1	p2	Result	Comment
1/8	2/8	TRUE	both displacements are in first half.
4/8	5/8	TRUE	both displacemants are in second half.
1/8	5/8	FALSE	p1 is in first half, p2 in second half.

Appendix 2 - Programs.

A2.1 Code for Parts Expert.

```
//                partsexp.h
#include "score.h"
const int MAXEXPERTSTORE = 50;

struct Limits      // stores lower and upper bar numbers of each 8 bar
part
{
 int lower, upper;
};

class PartsExpert
{
 Limits ar[MAXEXPERTSTORE];
 int size;                    // number of expanded 8-bar sections
 Score * sptr;

 public:

 PartsExpert( Score & s)
 {
  size = 0;
  sptr = &s;
  int n = 1;
  ScoreIterator si( s,0 );

  while (TRUE)
  {
   if (size + 1 == MAXEXPERTSTORE )
   {
    cerr << "\nno room in PartsExpert";
    return;
   }
   if ( !si.locate(BAR, n+7)) return;
   ar[size].lower = n;
   ar[size++].upper = n+7;
   if (si.step(BARLINE))
   {
    if (si.getBarType() < Set (CLHLC, CLLC, CLH, CLL, CLC, CL))
    {
     ar[size].lower = n;
     ar[size++].upper = n + 7;
    }
   }
   else return;
   n += 8;
  }
 }

 int isSingled()
 {
  // false for explicit repeats
  if ( ar[0].lower == ar[1].lower ) return FALSE;

  // must have at least two parts
  if ( size <= 2 ) return TRUE;

  // true if bars 1-8 dissimilar to bars 9-16
```

```
  ScoreIterator si1(*sptr,0), si2(*sptr,0);

  si1.locate(BAR,1);

  //d is distance of 7 1/2 bars + eight note
  Rat span = si1.getTimeSig() * Rat (15,2) + Rat(1,8);
  si2.locate(BAR,9);
  int diff = diff1( si1, si2, span );
  if ( diff > 300 ) return TRUE;

  // false otherwise
  return FALSE;
 }

 int numberOfParts()
 {
  if ( isSingled()) return size;
  return size/2;
 }

 int hasOddPart()
 {
  if ( !isSingled() && (size % 2) != 0 ) return TRUE;
  return FALSE;
 }

 int getBarNoForPart(int i)
 {
  int index;
  if ( isSingled() ) index = i - 1; else index = i*2 - 2;
  if ( index  < 0 || index >= size )
  {
   cerr <<
"\nPartsExpert::getBarForPart(int) called with impossible part number";
   cerr<< "\nParameter value is " << i << " giving index = " << index
       << ", parameter should be in range 0 to " << size;
   if ( isSingled() ) cerr << "( tune is singled )";
  }
  return ar[index].lower;
 }
};
//        end of partexp.h
```

A2.2 Program to Evaluate Average Pitches in Tune and Turn.

```
// To examine the average pitches in the first and second part of tunes
// includes provision for case where the the first part has been
// notated twice.

#define ___MAIN

#include <iostream.h>
#include <fstream.h>
#include "score.h"
#include "almain.h"
#include "score5.h"
#include "partsexp.h""

extern ofstream fout;

int main(int argc, char * argv[])
{

 if (argc != 2)
 {
  cout << "\nInvalid command line; should be  ex1 <filename> ";
  return 1;
 }

 long countTunes = 0;
 long countTunesRising = 0;

 String fname;

 while (getNextScoreNames(argv[argc-1], fname))
 {
  Score s(fname);
  if ( s.isNull() )
  {
   cout << "\nScore " << fname << " does not exist";
   return 2;
  }

  int sumOfPitches[2] = { 0, 0};
  int noteCountOfPart[2]    = { 0, 0};

  cout << "\n" << s.getString(TITLE) << " " << s.getString(ETITL);

  ScoreIterator si(s, 0);


  while ( si.getBarNo() <=8 && !si.isLast())
  {
   if ( si.getTag() == NOTE )
   {
    sumOfPitches[0] += si.getPitch12();
    noteCountOfPart[0]++;
   }
   si.step();
  }

  PartsExpert partsExpert(s);
```

```
  int nextBar = partsExpert.getBarNoForPart(2);

  while ( si.getBarNo() <= nextBar+8 && !si.isLast())
  {
   if ( si.getTag() == NOTE )
   {
    sumOfPitches[1] += si.getPitch12();
    noteCountOfPart[1]++;
   }
   si.step();
  }

  float average[2] = { 0, 0};

  for ( int count = 0; count < 2; count++ )
  {
   if ( noteCountOfPart[count] > 0 )
   {
    average[count] =
                  ((float)sumOfPitches[count])/ noteCountOfPart[count];
   }
   else
   {
    cerr << "\nno notes in tune!";
   }
  }
  countTunes++;
  if ( average[0] < average[1] ) countTunesRising++;
 }
 fout << "\n\n\nAverage pitches analysis of parts 1 and 2 of tunes.";
 fout << "\n\nFiles used from '" << argv[1] << "'."
      << "\n\nTunes with higher average pitch in 2nd part = "
      << countTunesRising << " out of " << countTunes << " ("
      << countTunesRising*100/countTunes << "%). ";

 return 0;
}
```

A2.3 Program to Search for the Occurrence of a Tuple.

```
 //                                    ex8.cpp

#define ___MAIN

#include <iostream.h>
#include <fstream.h>
#include "score.h"
#include "store.h"
#include "tuple.h"
extern ofstream fout;

void drawLine()
{
 fout <<
"--------------------------------------------------------------------------";
 return;
}

int main(int argc, char * argv[])
{

 if (argc < 2 || argc > 3)
 {
  cout<<"\nInvalid command line; should be  ex8 [-l<size>] <filename>";
  cout << "\nwhere size is the size of the tuple ( default 8).";
  return 1;
 }

 int tupleSize = 16;
 if ( argc == 3)
 {
  if ( *argv[1] == '-' && *(argv[1] +1) == 'l')
   sscanf(argv[1]+2,"%f",&tupleSize);
  else
  {
   cerr << "\nInvalid parameter " << argv[1] << "\n";
   return 1;
  }
 }

 cout << "\nSearch for pitch tuple";

 cout << "\nInput tuple of Size " << tupleSize;
 PitchTuple target(tupleSize);
 cout << '\n';

 for ( int i = 0; i < tupleSize; i++)
 {
  cout << i+1 << ':';
  int j;
  cin >> j;
  target.put(j, i);
 }
 cout << "\nSearching";
```

```
 int countAll = 0;

 String str, fname(argv[argc-1]);

 while (getNextScoreNames(fname, str))
 {
  Score s(str);
  ScoreIterator si(s, 0);
  countAll++;
  si.locate(BAR,1);
  PitchTuple tuple(tupleSize);
  int count = 0;

  while (si.getBarNo() != tupleSize/2 + 1 && ! si.isNullStave())
  {
   if ( si.getTag() == NOTE &&
      !(GRACE_NOTE < si.getAttributeSet()) &&
        ( si.getBarDist() == Rat(0,1) ||        // start of bar
          si.getBarDist()==(si.getTimeSig()/Rat(2))))// middle of bar
    tuple.put( si.getPitch12(), count++);
   si.step();
  }

  if ( target == tuple )
   cout << '\n' << si.getString(NUMBER) << ' ' << si.getString(TITLE)
        << ' ' << si.getString(ETITL);
 }
 return 0;
}
```

A2.4 Program used to Traverse and List the Entities in a Polyphonic Score.

```
Score s(filename);
ScoreIterator si(s);

printEntries(s, fout);
fout << '\n';

int more = TRUE;

while ( more )
{
 fout << '\n' << si.getStaveId() << ':'
      << si.getName() << ":" << si.getString();
 if ( si.isLast()) more = FALSE; else si.step();
}
```

Sample output of this program is shown in table A3.6 in appendix 3.

Appendix 3 - Output of Programs.

Program outputAccented Tone analysis for pieces in =d:\mdb\tdmoi\tdmoij.dir

Frequency	Pitch 16-Tuple
1	{0,12,16,9,7,7,2,2,0,12,16,9,7,12,0,0}
1	{0,12,14,12,0,12,4,2,0,12,14,12,0,12,16,12}
1	{0,12,14,11,16,17,14,5,0,12,14,14,16,17,14,12}
1	{0,12,9,7,5,4,2,7,0,12,9,7,5,2,7,12}
1	{0,12,9,4,0,4,-1,5,0,12,14,5,-5,2,4,0}
1	{0,12,7,9,0,12,7,17,0,12,7,9,19,21,7,2}
1	{0,12,7,7,0,12,2,2,0,12,7,12,9,5,0,0}
1	{0,12,7,4,0,12,5,5,0,4,2,0,16,7,2,5}
1	{0,12,7,2,-5,4,4,2,0,12,7,2,-5,4,2,0}
1	{0,11,4,5,0,2,7,2,0,11,12,5,0,0,0,0}
1	{0,9,12,9,0,9,10,7,5,9,12,9,0,7,9,5}
1	{0,9,12,9,0,9,4,9,0,9,12,9,0,7,9,5}
1	{0,9,12,9,0,9,4,7,0,9,12,9,17,12,7,5}
1	{0,9,10,7,9,5,7,4,0,9,10,7,12,0,9,5}
1	{0,9,10,4,9,7,5,7,0,9,10,4,9,7,0,5}
1	{0,9,10,4,0,9,10,5,0,9,10,4,16,16,10,5}
1	{0,9,9,9,0,12,2,2,0,9,9,9,0,0,4,0}
1	{0,9,7,9,0,9,7,2,0,9,7,9,9,10,9,2}
1	{0,9,7,5,14,12,7,9,0,9,7,5,14,12,7,5}
1	{0,9,7,4,0,9,7,10,0,9,7,4,4,4,7,10}
1	{0,9,0,0,0,9,0,2,0,9,0,-7,10,9,0,2}
1	{0,8,8,1,1,10,3,1,0,1,3,8,0,-2,-4,-4}
1	{0,8,5,8,0,8,8,5,0,8,5,8,13,8,8,5}
1	{0,8,5,8,0,8,5,0,0,8,5,8,0,8,5,-2}
1	{0,7,12,7,9,2,9,2,0,4,7,12,7,0,7,0}
1	{0,7,12,7,9,2,4,0,0,7,12,7,9,14,12,12}
1	{0,7,12,7,7,4,7,4,0,7,12,7,7,2,7,2}
1	{0,7,12,7,5,2,4,0,0,7,12,7,9,14,12,4}
1	{0,7,12,7,3,3,8,3,0,7,12,7,1,1,-4,1}
1	{0,7,11,11,7,12,16,11,4,5,11,11,7,7,4,0}
1	{0,7,8,7,0,7,1,5,0,7,8,7,12,7,1,-4}
1	{0,7,5,10,0,7,5,5,0,7,5,10,14,12,5,5}
1	{0,7,4,7,0,7,0,0,0,7,4,7,0,4,0,-3}
1	{0,7,4,4,0,7,7,10,0,7,4,4,9,10,7,5}
1	{0,7,2,7,0,7,2,4,0,7,4,7,12,7,2,4}
1	{0,7,2,5,0,7,5,0,0,7,2,5,3,7,5,0}
1	{0,7,0,3,7,5,7,2,0,7,0,3,5,10,5,-2}
1	{0,7,-2,5,0,7,3,12,15,10,5,2,0,5,7,0}
1	{0,6,8,10,12,6,1,1,0,6,8,10,12,6,3,8}
1	{0,5,12,12,7,5,7,0,0,5,12,12,9,10,5,5}
1	{0,5,12,10,9,12,17,7,0,5,12,10,9,12,17,5}
1	{0,5,12,8,7,-2,7,7,0,5,12,17,12,7,12,5}
1	{0,5,10,4,9,10,7,5,0,5,10,4,12,10,7,5}
1	{0,5,9,14,12,4,9,7,0,5,9,14,12,4,9,5}
1	{0,5,9,10,4,0,4,7,0,5,9,7,16,10,5,5}
1	{0,5,9,9,0,4,7,10,0,5,9,9,16,10,7,5}
1	{0,5,9,7,9,2,9,2,0,5,9,7,9,0,5,5}
1	{0,5,8,8,12,12,10,3,0,5,8,8,12,10,8,5}
1	{0,5,8,0,-2,-2,-2,3,0,5,8,0,0,5,0,-2}

1	{0,5,7,12,15,12,10,4,0,5,7,12,15,10,5,5}
1	{0,5,7,10,12,10,7,3,0,5,7,10,12,10,7,5}
1	{0,5,7,9,0,5,7,5,0,5,7,9,17,12,7,5}
1	{0,5,7,8,12,3,1,-2,0,5,7,8,12,3,5,8}
1	{0,5,7,8,7,0,3,-2,0,5,7,3,12,15,7,5}
1	{0,5,7,7,12,10,10,3,0,5,7,7,12,10,12,5}
1	{0,5,7,5,10,10,7,3,0,5,7,5,10,10,7,5}
1	{0,5,7,5,0,5,12,9,14,12,10,9,0,7,9,5}
1	{0,5,7,4,9,7,5,0,0,5,9,7,9,10,7,5}
1	{0,5,7,0,16,14,9,7,0,5,7,0,16,14,7,5}
1	{0,5,5,7,-2,3,3,5,0,5,7,7,12,10,8,5}
1	{0,5,4,10,9,14,9,7,0,5,4,10,9,14,9,5}
1	{0,5,4,7,9,10,9,4,0,5,4,7,9,10,7,5}
1	{0,5,3,3,0,5,5,0,0,5,3,3,8,7,0,-2}
1	{0,5,2,-3,-3,-3,-8,-5,0,5,2,-3,9,10,5,5}
1	{0,5,0,7,5,4,2,-1,0,5,0,7,5,4,11,12}
2	{0,5,0,0,0,5,9,2,0,5,0,0,5,7,9,5}
1	{0,5,0,0,0,5,2,2,0,5,5,9,10,5,-3,2}
1	{0,5,-3,3,-5,-2,-5,-2,0,5,-3,2,-2,3,0,-7}
1	{0,4,12,12,11,9,7,2,0,4,12,12,9,4,9,2}
1	{0,4,9,7,0,4,-3,2,0,4,9,7,2,4,0,0}
1	{0,4,7,12,9,7,-1,-1,0,4,7,12,9,7,-1,0}
1	{0,4,7,12,0,4,7,2,0,4,7,12,14,12,7,2}
1	{0,4,7,12,0,4,2,5,0,4,7,12,9,7,5,0}
1	{0,4,7,4,2,5,9,7,0,4,7,4,2,5,4,-3}
1	{0,4,7,4,2,2,9,2,0,4,7,4,2,4,0,-3}
1	{0,4,7,4,0,4,7,2,0,4,7,12,11,9,7,2}
1	{0,4,7,2,-2,2,5,5,0,4,7,2,7,0,0,0}
1	{0,4,5,9,0,4,5,2,7,0,5,9,7,5,0,12}
1	{0,4,5,4,0,5,-1,-5,0,4,5,4,7,5,0,0}
1	{0,4,4,7,9,7,9,7,0,4,4,7,9,7,2,2}
1	{0,4,4,7,5,7,4,2,0,4,4,7,5,7,4,0}
1	{0,4,4,4,0,4,2,-3,0,4,4,4,9,4,4,-3}
1	{0,4,4,-1,-3,7,4,7,0,4,4,2,9,2,-3,-3}
1	{0,4,2,5,0,7,12,7,5,4,2,5,7,5,0,0}
1	{0,4,2,5,0,4,7,9,0,4,2,5,7,10,5,-2}
1	{0,4,2,5,0,4,7,9,0,4,2,5,0,7,12,5}
1	{0,4,2,0,7,-1,-1,-5,0,4,2,11,5,-1,4,0}
1	{0,4,0,4,-1,2,7,2,0,4,0,4,7,2,0,-3}
1	{0,4,0,4,-3,-7,-3,-2,0,4,0,0,9,7,5,5}
1	{0,4,0,-3,0,4,-1,-5,0,4,0,4,0,-5,0,-3}
1	{0,4,0,-3,-1,2,-1,-5,0,5,0,-3,0,0,5,0}
1	{0,3,8,10,12,13,8,10,0,3,8,10,12,13,3,8}
1	{0,3,8,10,0,3,8,1,0,3,8,10,12,13,3,8}
1	{0,3,8,8,3,1,5,-2,0,3,8,8,3,5,0,-4}
1	{0,3,8,6,0,6,0,3,0,3,8,6,0,1,0,1}
1	{0,3,8,5,0,3,12,8,0,3,8,5,0,3,12,8}
1	{0,3,8,3,1,3,8,7,0,3,8,3,1,3,3,-4}
1	{0,3,8,0,1,0,1,-2,0,3,8,0,-2,7,8,-4}
1	{0,3,7,10,7,7,3,5,0,3,7,10,7,2,3,0}
1	{0,3,7,10,2,-2,2,2,0,3,7,10,14,12,14,12}
1	{0,3,7,7,12,3,7,3,-2,3,7,7,12,15,7,5}
1	{0,3,7,7,-2,2,5,5,0,3,7,7,12,8,7,0}

1	{0,3,7,3,-2,2,5,5,0,3,7,15,10,3,3,0}
1	{0,3,7,0,0,3,7,0,0,3,7,2,-2,10,5,-2}
1	{0,3,5,7,0,3,5,2,0,3,5,7,12,8,7,2}
1	{0,3,5,6,5,10,3,0,1,3,5,6,5,10,3,1}
1	{0,3,5,3,3,3,3,3,0,3,5,3,-4,1,-4,1}
2	{0,3,5,3,0,3,-2,-2,0,3,5,3,0,1,-4,-4}
1	{0,3,3,8,1,0,-2,-2,0,3,3,8,3,8,3,-4}
1	{0,3,3,3,5,3,5,-2,0,3,3,8,5,3,3,-4}
1	{0,3,3,3,1,0,5,-2,0,3,3,8,5,3,3,-4}
1	{0,3,3,1,-2,7,8,7,0,3,3,1,-4,8,0,1}
1	{0,3,2,0,0,3,5,-2,0,3,2,0,10,10,5,-2}
1	{0,3,1,-5,0,0,1,-4,0,3,1,-5,3,8,1,-4}
1	{0,3,0,3,1,5,1,5,0,3,0,3,-4,5,8,-2}
1	{0,3,0,-5,-9,-5,-2,3,0,0,3,8,7,-2,-5,-7}
1	{0,3,-5,3,2,0,-2,2,0,3,-5,3,2,3,-2,2}
1	{0,2,9,2,0,2,9,-3,0,2,9,2,7,12,5,0}
1	{0,2,5,7,0,2,-1,-5,0,2,5,7,-3,2,4,0}
1	{0,2,5,5,0,2,2,2,0,2,5,5,7,2,0,0}
1	{0,2,5,4,0,2,-3,2,0,2,4,12,7,2,-5,0}
1	{0,2,4,9,4,0,0,-3,0,2,4,9,4,0,2,0}
1	{0,2,4,9,0,2,4,-3,0,2,4,9,7,2,4,0}
1	{0,2,4,7,12,7,5,2,0,5,4,7,12,7,4,0}
1	{0,2,4,7,4,0,2,0,0,2,4,7,4,0,2,0}
1	{0,2,4,7,0,2,4,-3,0,2,4,7,5,4,0,0}
1	{0,2,4,5,0,2,-5,-1,0,2,4,5,4,2,-5,0}
1	{0,2,4,2,0,2,0,0,0,2,0,5,7,2,0,0}
1	{0,2,4,-5,0,2,-5,-5,0,2,4,-3,2,-5,-10,-10}
1	{0,2,3,5,-2,-9,-5,-5,0,-10,-5,3,-2,3,-5,-7}
1	{0,2,2,9,0,2,2,5,0,2,2,9,14,12,14,9}
1	{0,2,0,7,9,12,17,14,0,2,0,7,9,12,16,12}
1	{0,2,0,7,0,2,0,-5,0,2,0,7,0,0,2,5}
1	{0,2,0,7,0,2,0,-5,0,2,0,7,0,0,-3,-7}
1	{0,2,0,4,9,7,2,2,0,2,0,4,9,7,2,0}
1	{0,2,0,2,0,3,-2,-9,0,2,0,2,5,3,-2,-7}
1	{0,2,0,-3,0,2,0,-5,0,2,0,-3,-5,-2,0,-5}
1	{0,2,0,-3,0,2,-1,-5,0,2,4,9,4,-1,0,-3}
1	{0,2,0,-5,-7,-8,-7,-10,0,2,0,-5,-3,2,4,0}
1	{0,2,-5,0,9,7,4,2,0,2,-5,0,9,7,4,0}
1	{0,2,-5,-1,-10,-10,-17,-10,0,2,-5,-1,-10,-10,-17,-12}
1	{0,2,-5,-10,-8,-5,-3,2,0,2,-5,-10,-8,-5,-3,0}
1	{0,2,-7,-7,-12,-7,-3,-5,0,0,-7,-7,-12,-5,-3,-7}
1	{0,1,3,7,3,8,3,8,0,1,3,7,1,3,1,3}
1	{0,1,3,1,0,1,3,8,0,1,3,3,12,7,3,8}
1	{0,1,3,-2,-4,5,7,3,8,8,1,3,-4,1,0,-4}
1	{0,1,3,-4,0,0,-5,-5,0,1,3,-2,0,-2,-9,-4}
1	{0,1,0,1,0,1,0,-4,0,1,0,3,0,1,0,-4}
1	{0,1,0,1,0,-2,-7,-7,0,1,0,1,0,-2,-9,-9}
1	{0,1,-4,0,-9,-4,5,1,0,1,-4,0,-9,-4,0,5}
1	{0,0,12,9,9,7,2,2,0,0,12,9,9,7,2,0}
1	{0,0,12,7,4,7,-1,2,0,0,12,7,4,5,0,0}
1	{0,0,12,0,4,5,0,-1,0,0,12,0,4,5,4,0}
1	{0,0,9,7,0,0,9,2,0,0,9,12,9,2,4,0}
1	{0,0,7,12,4,0,7,-1,0,0,7,12,4,9,0,0}

1 {0,0,7,12,0,0,2,2,0,0,7,12,9,7,5,0}
1 {0,0,7,12,0,0,2,2,0,0,7,12,9,7,-1,0}
1 {0,0,7,10,-2,2,5,0,0,0,7,5,10,9,2,0}
1 {0,0,7,9,0,0,-5,-5,0,0,7,9,0,9,5,5}
1 {0,0,7,7,10,10,5,2,0,0,7,7,10,5,0,0}
1 {0,0,7,7,9,7,5,5,0,0,7,7,9,7,5,0}
1 {0,0,7,7,9,7,2,4,0,0,7,7,9,7,2,0}
1 {0,0,7,7,9,7,2,2,0,0,7,7,9,7,0,0}
1 {0,0,7,5,9,7,2,2,0,12,7,5,9,7,0,0}
1 {0,0,7,5,3,0,7,1,0,0,7,7,12,7,-2,-4}
1 {0,0,7,5,0,0,7,1,0,0,7,7,12,8,7,1}
1 {0,0,7,5,-2,2,5,2,0,0,7,10,7,2,7,0}
1 {0,0,7,4,0,0,7,4,0,0,7,4,7,-2,-3,-7}
1 {0,0,7,3,7,12,10,15,12,17,10,3,7,10,7,5}
1 {0,0,7,2,4,5,4,12,0,0,7,2,4,5,4,0}
1 {0,0,7,0,-2,-2,5,-2,0,0,7,10,7,2,7,0}
1 {0,0,5,5,2,5,-2,-5,0,5,2,7,12,7,9,5}
1 {0,0,5,0,0,0,5,-2,0,0,5,0,5,5,0,-4}
1 {0,0,4,9,0,0,2,2,0,0,4,7,9,9,2,-5}
1 {0,0,4,7,0,0,5,4,0,0,4,7,0,0,4,0}
1 {0,0,4,7,0,0,2,7,0,0,4,2,7,11,2,5}
1 {0,0,2,2,9,12,2,2,0,0,2,2,9,12,5,0}
1 {0,0,2,0,-2,-3,-2,-8,0,2,5,4,4,-2,-5,-7}
1 {0,0,2,-5,0,0,2,5,0,0,2,-5,-1,-1,2,5}
1 {0,0,2,-5,0,-3,-1,-8,0,0,2,-5,0,-5,-3,-3}
1 {0,0,1,-2,1,-4,3,8,0,0,1,-2,0,1,-2,-4}
1 {0,0,1,-4,0,0,3,1,0,0,3,12,8,6,3,1}
1 {0,0,0,10,12,4,9,7,0,0,0,10,12,4,9,5}
1 {0,0,0,10,0,0,7,2,0,0,0,10,5,10,7,2}
1 {0,0,0,7,0,0,-1,2,0,0,0,7,5,4,-1,2}
1 {0,0,0,5,12,7,9,2,0,0,0,5,12,7,9,5}
1 {0,0,0,3,0,0,-2,-5,0,2,3,2,0,3,-2,-5}
1 {0,0,0,0,3,8,-2,-4,0,0,0,0,3,8,-4,-4}
1 {0,0,0,0,2,2,5,9,7,14,12,5,9,9,5,2}
1 {0,0,0,0,1,0,-2,-2,0,0,0,0,8,3,-4,-4}
1 {0,0,0,0,0,0,-2,-2,0,0,0,0,5,0,-2,-2}
1 {0,0,0,0,0,0,-2,-5,0,0,0,0,-2,3,0,-5}
1 {0,0,0,0,0,0,-2,-7,0,0,0,0,-2,3,-2,-7}
1 {0,0,0,-3,-2,2,-3,0,0,0,0,-3,-2,2,3,-2}
1 {0,0,0,-7,0,0,2,-5,0,0,0,0,2,0,2,-5}
1 {0,0,0,-7,-2,-2,-2,-9,-3,-2,0,5,5,0,-3,-7}
1 {0,0,-1,-1,0,7,2,-5,0,0,-1,-1,4,0,-7,-12}
1 {0,0,-1,-1,0,0,2,5,0,0,-1,-1,5,-1,2,0}
1 {0,0,-1,-5,0,0,2,-5,0,0,-1,-5,0,-2,-7,-12}
1 {0,0,-2,3,0,0,-5,-2,0,0,-2,3,-9,1,-2,-4}
1 {0,0,-2,3,-2,-7,-2,-7,0,0,-2,3,-2,-9,-2,-9}
1 {0,0,-2,-2,3,1,-4,8,0,0,-2,-2,3,1,-4,-4}
1 {0,0,-2,-2,-7,0,-2,-2,0,4,-2,-2,-7,3,2,-2}
1 {0,0,-2,-5,-4,-2,0,0,0,0,-2,-5,-9,3,-5,-7}
1 {0,0,-2,-5,-9,-5,-5,-7,0,0,-2,-5,-9,-5,-7,-9}
1 {0,0,-2,-7,-9,-9,-5,-9,2,0,5,0,3,-2,0,-7}
1 {0,0,-3,0,5,0,-2,-5,0,0,-3,0,5,-5,-3,-7}
1 {0,0,-3,-3,0,0,-7,5,0,0,-3,-3,-7,-3,-7,5}

1	{0,0,-4,8,0,0,0,1,0,0,-4,8,0,0,0,-4}
1	{0,0,-4,8,0,0,-2,1,0,0,-4,8,1,0,-2,1}
1	{0,0,-4,8,-2,-2,0,3,0,0,-4,8,5,7,7,3}
1	{0,0,-4,1,5,1,6,0,5,5,6,3,8,6,3,1}
1	{0,0,-4,0,0,0,-2,1,0,0,-4,0,5,7,-2,1}
1	{0,0,-4,-4,5,3,0,-2,0,0,-4,-4,5,3,0,-4}
1	{0,0,-5,-2,0,0,3,7,0,0,-5,-2,3,10,3,0}
1	{0,0,-5,-5,0,0,-10,2,0,0,-5,-5,4,0,-3,0}
1	{0,0,-5,-7,-19,-7,-2,-5,0,0,-5,-7,-19,-7,-2,-7}
1	{0,0,-5,-9,0,0,-2,3,0,0,-5,-9,-2,3,-5,-7}
1	{0,0,-7,-5,-8,-7,-8,-1,0,0,-7,-5,-8,-7,-8,-12}
1	{0,0,-7,-7,-3,2,-3,-5,0,0,-7,-7,-2,-3,-12,-17}
1	{0,0,-8,-5,0,0,-8,-3,0,0,-8,-5,0,5,0,-3}
1	{0,0,-8,-12,0,0,-3,2,0,0,-8,-12,0,4,0,-3}
1	{0,-1,0,-3,-5,7,-1,-5,0,2,4,12,9,4,0,-3}
1	{0,-1,0,-8,0,-1,-5,-10,0,-1,0,-8,-12,-12,-5,-10}
1	{0,-1,-3,-1,-10,-5,-3,-1,0,-1,-3,-1,-10,-3,-1,-5}
1	{0,-1,-3,-5,0,2,-8,-3,0,-1,-3,-5,-8,-8,-8,-3}
1	{0,-1,-3,-5,0,-1,-1,-3,0,-1,-3,-5,0,2,0,-3}
1	{0,-1,-3,-8,0,-1,0,-10,0,-3,-5,-8,-12,-12,-5,-10}
1	{0,-1,-8,0,0,-1,-3,-5,0,-1,-8,0,0,-1,-3,-5}
1	{0,-1,-8,-12,-10,-1,0,-1,-5,-1,-8,-12,-10,-5,-8,-12}
1	{0,-2,7,3,0,0,7,1,0,0,7,7,12,8,7,1}
1	{0,-2,0,-4,-7,-2,-7,-2,0,-2,0,-4,-7,-4,-7,-4}
1	{0,-2,-2,-2,0,3,7,7,7,5,2,2,7,2,2,0}
1	{0,-2,-4,-4,5,8,0,-2,0,-2,-4,-4,5,8,1,-4}
1	{0,-2,-4,-4,5,8,0,-2,0,-2,-4,-4,5,7,8,8}
1	{0,-2,-4,-5,0,1,3,8,0,-2,-4,-5,0,-5,-9,-4}
1	{0,-2,-4,-5,0,1,-9,-4,0,-2,-9,-5,7,1,-9,-4}
1	{0,-2,-4,-5,0,-2,-4,7,0,-2,-4,-5,0,-5,-4,-4}
1	{0,-2,-4,-9,0,-2,-4,-11,0,-2,0,-2,0,1,-4,-11}
1	{0,-2,-4,-9,-11,-4,-11,-14,0,-2,-4,-9,-11,-4,-12,-16}
1	{0,-2,-5,-9,0,-2,-5,-7,0,-2,-5,-9,3,-2,-5,-9}
1	{0,-2,-9,0,0,3,0,-2,0,-2,-9,0,0,3,-2,-4}
1	{0,-2,-9,-2,-9,-2,0,5,0,-2,-9,-2,-9,-2,0,-4}
1	{0,-2,-9,-4,0,1,3,7,0,-2,-9,-5,0,1,-2,-4}
1	{0,-2,-9,-5,0,0,1,-5,0,-2,-4,8,7,8,1,-4}
1	{0,-2,-9,-9,-4,1,3,-2,0,-2,-9,-9,0,-2,-4,-4}
1	{0,-3,4,2,-3,0,4,-2,-3,5,10,4,9,4,-2,-7}
1	{0,-3,4,-3,-1,-5,2,-5,12,7,4,7,2,-5,0,-3}
1	{0,-3,0,-3,0,-3,-7,-10,0,-3,0,-3,-5,-3,-7,-10}
1	{0,-3,0,-7,0,-3,-8,-5,0,-3,0,2,-3,0,5,-7}
1	{0,-3,-2,-5,-12,-8,-7,-5,0,-3,-2,-5,-12,-8,-3,-7}
1	{0,-3,-3,-3,-1,-1,-1,-5,0,-3,-3,0,4,2,0,-3}
1	{0,-3,-3,-5,-7,-7,-3,-7,0,-3,-3,-5,-7,-7,-3,-7}
1	{0,-3,-5,-10,-17,-10,-8,-10,0,-3,-5,-10,-17,-10,-8,-12}
1	{0,-3,-7,5,9,0,-8,-5,0,-3,-7,5,7,0,-3,-7}
1	{0,-3,-8,0,-1,-5,-1,7,0,-3,0,9,7,-1,0,-3}
1	{0,-3,-8,-3,-1,-5,2,-5,0,-3,-8,9,9,4,0,-3}
1	{0,-3,-8,-5,-8,-5,-8,-10,0,-3,-8,-5,-8,-5,-8,-12}
1	{0,-3,-8,-12,-8,-8,-7,-10,0,-3,-8,-12,-7,-10,-8,-12}
1	{0,-4,8,3,0,-4,1,-2,0,-4,8,3,0,-2,-4,-4}
1	{0,-4,8,3,-2,-5,7,-5,0,-4,8,3,1,-2,3,-4}

1 {0,-4,8,3,-2,-5,-2,-5,0,-4,8,3,1,-2,3,-4}
1 {0,-4,3,0,0,-4,3,5,0,-4,3,0,-2,8,1,1}
1 {0,-4,3,-4,-2,-6,1,-2,0,-4,3,-4,1,-2,3,-4}
1 {0,-4,3,-5,-9,-11,-12,-11,-12,-16,-9,-5,-12,-11,-16,-16}
1 {0,-4,1,6,8,8,10,3,0,-2,3,6,8,1,-4,-4}
1 {0,-4,1,3,1,3,-5,-2,0,3,8,3,5,3,-2,-4}
1 {0,-4,1,-2,0,-4,1,-5,0,-4,1,-2,3,1,-2,-4}
1 {0,-4,0,2,0,-4,-2,-5,0,-4,0,-5,-9,3,-2,-5}
1 {0,-4,0,-5,-7,-4,0,0,0,-4,0,-5,-7,3,0,-7}
1 {0,-4,-2,-4,0,-4,-4,5,0,-4,-2,-4,-2,-5,-5,1}
1 {0,-4,-2,-5,-2,7,1,-5,0,-4,-2,-5,-2,7,1,-4}
1 {0,-4,-4,8,1,0,5,-2,0,-4,-4,8,1,-2,0,-4}
1 {0,-4,-4,7,0,-4,3,1,0,-4,-4,7,0,1,3,1}
1 {0,-4,-4,5,0,-4,5,3,8,7,5,3,0,-9,-4,-4}
1 {0,-4,-4,1,3,7,3,8,0,-4,-4,1,3,7,8,8}
1 {0,-4,-4,-2,0,-4,1,-2,0,-4,-4,-2,0,3,1,-4}
1 {0,-4,-4,-2,0,-4,1,-2,0,-4,-4,-2,-4,6,1,-2}
1 {0,-4,-4,-4,1,-4,-2,3,0,-4,-4,-4,8,3,-2,1}
1 {0,-4,-5,-5,-5,-11,-9,0,1,-4,-5,-5,-5,-11,-9,-4}
1 {0,-4,-7,-4,0,3,0,-2,0,-4,-7,-4,0,-2,0,-4}
1 {0,-4,-7,-4,0,0,-4,1,0,-4,-7,-4,5,0,-4,1}
1 {0,-4,-7,-4,0,-4,5,-2,0,-4,-9,-4,3,3,0,-4}
1 {0,-4,-7,-4,0,-4,-7,-2,0,-4,-7,-4,3,3,0,-4}
1 {0,-4,-9,-4,0,3,-2,-2,0,-4,-9,-4,0,3,-9,-4}
1 {0,-4,-9,-5,-4,-4,1,-5,0,0,-2,-2,3,-2,-4,-4}
1 {0,-5,2,4,0,-3,0,-3,0,-5,2,4,4,0,4,0}
1 {0,-5,0,9,12,4,5,-3,0,-5,0,9,12,4,5,0}
1 {0,-5,0,5,7,2,4,-3,0,-5,0,5,7,2,4,0}
1 {0,-5,0,-5,0,4,2,-3,0,-5,0,-5,0,7,4,0}
1 {0,-5,0,-5,0,-5,0,5,0,-5,0,-5,-7,-10,-1,-5}
1 {0,-5,0,-5,0,-5,0,-6,0,-5,0,-3,-6,6,0,-5}
1 {0,-5,0,-5,0,-5,-2,-5,0,-5,0,-5,3,-2,-5,-2}
1 {0,-5,0,-5,-3,-3,-3,-5,0,-5,0,-5,-5,2,4,0}
1 {0,-5,0,-5,-12,-9,-12,-9,0,-5,0,-5,-12,-11,-12,-16}
1 {0,-5,0,-7,0,-5,-2,-9,0,-5,0,-7,-9,-5,-2,-7}
1 {0,-5,-2,-5,0,-5,-7,-4,0,-5,-5,-2,0,1,-4,-7}
1 {0,-5,-2,-5,0,-5,-12,-7,0,-5,-2,-5,0,4,5,-7}
1 {0,-5,-2,-9,0,-5,-4,-11,0,-5,-4,-2,3,-2,-5,-9}
1 {0,-5,-3,-3,0,-5,-3,6,0,-5,-3,-3,0,-5,-3,2}
1 {0,-5,-3,-7,0,-5,2,2,0,-5,-3,-7,9,7,2,2}
1 {0,-5,-5,2,0,-5,0,0,0,-5,-5,2,7,2,0,0}
1 {0,-5,-5,-5,0,-5,-7,-7,0,-5,-5,-5,0,-5,0,-7}
1 {0,-5,-7,-2,0,-3,-5,2,0,-5,-7,5,0,-5,-7,-7}
1 {0,-5,-7,-8,0,-5,-8,-10,0,-5,-8,0,0,-5,-8,-12}
1 {0,-5,-7,-9,-12,-4,1,-2,0,-5,-7,-9,-12,-4,1,-4}
1 {0,-5,-7,-12,-5,-5,-3,-5,0,-5,-7,-12,-5,-5,-3,-7}
1 {0,-5,-7,-12,-7,-1,-7,-2,0,-5,-7,-12,-7,-2,-12,-12}
1 {0,-5,-7,-12,-10,-12,-17,-17,0,-5,-7,-12,-15,-17,-19,-19}
1 {0,-5,-8,-3,4,2,0,-5,0,-5,-5,-5,4,2,-1,-3}
1 {0,-5,-8,-3,0,4,-1,-5,7,4,2,-1,4,-5,-8,-3}
1 {0,-5,-8,-8,0,-5,0,-7,0,-5,-8,-8,0,-5,0,-7}
1 {0,-5,-9,-2,0,-2,-5,-2,0,-5,-9,-2,0,-2,-4,-7}
1 {0,-5,-9,-5,-12,-16,-12,-11,-9,-5,-9,-9,0,-2,-4,-4}

```
1    {0,-5,-9,-9,-5,-5,-5,-2,-2,-7,-14,-9,-5,-5,-5,-9}
1    {0,-5,-10,-5,0,-5,-8,-3,0,-5,-10,-5,0,5,0,-3}
1    {0,-5,-10,-7,-3,-3,-5,-10,0,-5,-10,-7,-3,-3,-5,-12}
1    {0,-5,-12,-5,0,-5,-10,-3,0,-5,-12,-5,0,-5,-10,-3}
1    {0,-5,-12,-5,-3,-7,-7,-7,0,-5,-12,-5,-3,2,4,0}
1    {0,-6,-13,-13,-13,-5,0,-3,-1,-6,-6,-13,-17,-3,-1,-5}
1    {0,-7,5,4,2,-5,2,2,0,-7,5,4,2,7,5,5}
1    {0,-7,3,0,-5,-9,-5,0,0,-7,0,5,8,7,0,5}
1    {0,-7,2,2,2,0,-2,-5,0,-7,2,2,2,-3,-3,-7}
1    {0,-7,2,-3,0,-7,-2,-5,0,-7,2,-3,2,-3,-3,-7}
1    {0,-7,0,5,0,-7,-5,2,0,-7,0,5,2,4,5,5}
1    {0,-7,0,3,-2,-9,-5,-5,0,-7,0,3,5,-2,-7,-7}
1    {0,-7,0,3,-2,-9,-9,3,0,-7,0,3,5,-2,-5,-7}
1    {0,-7,0,2,0,-7,9,2,0,-7,0,2,5,10,9,5}
1    {0,-7,0,0,2,5,2,2,0,-7,0,0,5,10,9,2}
1    {0,-7,0,-2,0,-7,0,-5,0,-7,0,-2,-7,2,0,-5}
1    {0,-7,0,-2,0,-7,-5,-7,0,-7,0,-2,0,5,-5,-7}
1    {0,-7,0,-3,2,10,4,0,0,-3,2,0,5,10,9,5}
1    {0,-7,0,-5,0,-7,-2,-5,0,3,0,-5,0,3,-2,-9}
1    {0,-7,0,-7,0,0,-5,-5,0,-7,0,-7,0,5,-7,-7}
1    {0,-7,0,-7,0,-7,-2,-5,0,-7,0,5,9,2,0,-7}
1    {0,-7,0,-10,-12,-7,0,-5,0,-7,-10,-12,-2,0,-5,-7}
1    {0,-7,-2,-5,0,-7,-5,-2,0,-7,-2,-5,-3,-2,-3,-7}
1    {0,-7,-2,-8,-3,2,4,-2,0,-7,-2,-5,-3,-2,-5,-7}
1    {0,-7,-2,-9,0,-4,1,0,0,-7,-2,-9,-7,0,-4,-4}
1    {0,-7,-2,-9,-3,0,-2,3,0,-7,-2,-9,-3,0,-5,-7}
1    {0,-7,-2,-9,-12,-4,1,5,0,-7,-2,-9,-12,-4,1,-4}
1    {0,-7,-4,-7,0,-7,-2,-9,-4,-7,0,-7,2,0,-2,-7}
1    {0,-7,-5,-12,0,-7,2,4,0,-7,-5,-12,-10,-5,-7,-7}
1    {0,-7,-5,-12,-10,-5,-3,-8,0,-7,-5,-12,-10,-5,-7,-7}
1    {0,-7,-7,2,2,0,2,-5,0,-7,-7,2,2,0,-3,-7}
1    {0,-7,-7,2,0,-7,2,-5,0,-10,-12,-5,0,-3,-3,-7}
1    {0,-7,-7,-7,0,-7,-7,-2,0,-7,-7,-7,0,-7,-7,-4}
1    {0,-7,-7,-7,-2,-9,-9,3,0,-7,-7,3,7,-2,-4,-7}
1    {0,-7,-9,3,1,3,0,-2,0,3,1,3,1,0,0,-4}
1    {0,-8,0,0,0,-8,-3,-10,0,-8,0,0,2,0,-5,-5}
1    {0,-8,0,0,0,-8,-3,-10,0,-8,0,-1,4,2,-3,-10}
1    {0,-8,0,0,0,-8,-5,-10,0,-8,0,0,2,-3,-10,-5}
1    {0,-8,-5,-5,0,-8,-5,5,0,-8,-5,-5,-10,-10,-3,-3}
1    {0,-8,-5,-5,-7,-8,-10,-5,0,-8,-5,-5,-7,-8,-10,-12}
1    {0,-8,-5,-10,0,-8,0,2,-5,-10,-10,-10,-8,-5,0,-1}
1    {0,-8,-7,-8,0,-8,-3,-10,0,-8,-7,-8,-12,0,-5,-12}
1    {0,-8,-8,-8,-5,-10,-10,-10,0,-8,-12,0,4,-1,-5,-5}
1    {0,-8,-12,-8,-5,-10,-10,-10,0,-8,-12,0,4,0,-3,-5}
1    {0,-9,0,3,1,-5,1,5,0,-9,0,3,1,-2,0,-4}
1    {0,-9,-5,-12,0,-9,-7,-14,0,-9,-7,-5,0,-2,-7,-12}
1    {0,-9,-14,-9,-14,-4,-5,-12,0,-9,-14,-9,-7,-4,-5,-9}
1    {0,-12,-3,-1,0,2,7,-1,0,-12,-3,-1,0,2,4,0}
1    {0,-12,-15,-17,0,-12,-15,-13,0,-12,-15,-17,-15,-13,-8,-12}

Total number of pieces processed is 365
```

Table A3.1 Frequency distribution of tuples for TDMOI using program of Ex.4.

```
Analysis of Initial Notes
File: =d:\mdb\tdmoi\djig.dir
365 scores processed

NIEPrimes/Tuples          Frequency
=================================================================
Nr. of Notes:1            61   16%
_________________________________________________________________
NIEPF:{0 } 1              61   16%
_________________________________________________________________
Tuple:{0}                 61   16%
=================================================================
Nr. of Notes:2            181  49%
_________________________________________________________________
NIEPF:{0 5 } 2-5          64   17%
NIEPF:{0 4 } 2-4          3    0%
NIEPF:{0 3 } 2-3          12   3%
NIEPF:{0 2 } 2-2          46   12%
NIEPF:{0 1 } 2-1          31   8%
NIEPF:{0 } 1              25   6%
_________________________________________________________________
Tuple:{0,9}               1    0%
Tuple:{0,7}               2    0%
Tuple:{0,5}               53   14%
Tuple:{0,4}               1    0%
Tuple:{0,3}               2    0%
Tuple:{0,2}               22   6%
Tuple:{0,1}               5    1%
Tuple:{0,0}               23   6%
Tuple:{0,-1}              26   7%
Tuple:{0,-2}              24   6%
Tuple:{0,-3}              9    2%
Tuple:{0,-4}              2    0%
Tuple:{0,-5}              8    2%
Tuple:{0,-7}              1    0%
Tuple:{0,-12}             2    0%
=================================================================
Nr. of Notes:3            112  30%
_________________________________________________________________
NIEPF:{0 4 7 } I-3-11     3    0%
NIEPF:{0 4 5 } I-3-4      3    0%
NIEPF:{0 3 5 } I-3-7      7    1%
NIEPF:{0 2 5 } 3-7        9    2%
NIEPF:{0 2 4 } 3-6        20   5%
NIEPF:{0 2 3 } I-3-2      21   5%
NIEPF:{0 1 3 } 3-2        48   13%
NIEPF:{0 2 } 2-2          1    0%
_________________________________________________________________
Tuple:{0,2,4}             13   3%
Tuple:{0,2,3}             13   3%
Tuple:{0,2,0}             1    0%
Tuple:{0,1,3}             10   2%
```

```
Tuple:{0,-1,-3}           8    2%
Tuple:{0,-1,-5}           3    0%
Tuple:{0,-2,-3}           37   10%
Tuple:{0,-2,-4}           7    1%
Tuple:{0,-2,-5}           7    1%
Tuple:{0,-3,-2}           1    0%
Tuple:{0,-3,-5}           9    2%
Tuple:{0,-3,-7}           3    0%
=================================================================
Nr. of Notes:4            11   3%
_________________________________________________________________
NIEPF:{0 2 4 5 } I-4-11   10   2%
NIEPF:{0 1 3 5 } 4-11     1    0%
_________________________________________________________________
Tuple:{0,2,4,5}           10   2%
Tuple:{0,-2,-4,-5}        1    0%
=================================================================
```

Table A3.2 Initial anacrusis details for TDMOI.

```
Calculation of forms for file =d:\mdb\tdmoi\djig.dir
Key transitions processed
Stresses processed
Critical Value = 40


Form        Frequency

abcd efgh   20
abcd efgg   1
abcd efgd   3
abcd efgb   2
abcd efdg   1
abcd ecfg   1
abcd ebfg   10
abcd ebcf   3
abcd ebcb   1
abcd defg   1
abcd cefg   3
abcd cefd   1
abcd befg   1
abcd aefg   37
abcd aefe   1
abcd aefd   3
abcd aecf   1
abcd aeaf   1
abcd abef   68
abcd abee   1
abcd abed   11
abcd abec   1
abcd abeb   1
abcd abce   46
abcd abcd   1
abcd abae   2
abcd aaef   2
abcc adee   1
abcc abde   3
abcc abdd   1
abcc abcd   1
abcb defg   1
abcb decf   1
abcb abde   2
abca dbef   1
abac defg   4
abac defc   1
abac dcef   1
abac dbef   4
abac dbec   1
abac bdef   1
abac adef   27
abac adec   7
abac adeb   1
```

```
abac adae   2
abac abde   42
abac abdc   11
abac abdb   2
abac abcd   1
abac abad   3
abac aadb   1
abab cdef   1
abab acde   3
abab acdb   2
abab abcd   1
aabc adce   1
aabc aade   9
aabc aadc   1
aabc aabd   2
aaba aabc   1
```

Table A3.3 Frequency distribution of form for tune parts of double jigs in TDMOI.

```
Calculation of distances for files =d:\mdb\tdmoi\djig.dir and
                                   =d:\mdb\tdmoi\djig1.dir
Key transitions processed
Stresses processed
Window widths processed

(6)an doctuir ua neill - DOCTOR O'NEILL
    (224)brighidin ni mhaoldomhnaigh - BIDDY MALONEY  4=7 (88.2:0)
(10)rogha ui gadhra - GUIRY'S FAVOURITE
    (221)briain ua floinn - BRYAN O'LYNN  1=2 (95.8:0)
(11)bean-cheile ui maoileoin - MALOWNEY'S WIFE
    (224)brighidin ni mhaoldomhnaigh - BIDDY MALONEY  1=1 (55.2:0)
      2=2 (96.5:0)      3=3 (64.6:0)
(13)sugra bheantraighe - THE HUMOURS OF BANTRY
    (98)an abhraiseach - THE FLAXDRESSER  1=3 (95.8:0)
(15)an bothar go bhaile-atha-chliath - THE HIGHWAY TO DUBLIN
    (293)an mor ata aci? - HOW MUCH HAS SHE GOT?      1=1 (75:0)
(16)ann do tinneas ne tae ta uait? - WHEN SICK IS IT TEA YOU WANT?
    (358)inthigh do'n diabhal's corruidh tu fein - GO TO THE DEVIL
      AND SHAKE YOURSELF      1=1 (5.6:0)       2=2 (0:0)
(29)alltri na mna - CHERRISH THE LADIES
    (56)sugra an cheapaigh - THE HUMOURS OF CAPPA     2=2 (93.1:0)
(34)tomas ua gaillimh - GALWAY TOM
    (144)an teach annsa gleann - THE HOUSE IN THE GLEN      7=2
      (75.3:0)
    (199)an bho bhreach - THE SPOTTED COW 7=1 (87.2:0)
(39)cuairt go h-eirinn - A VISIT TO IRELAND
    (147)carabhat mhic sheoin - JACKSON'S CRAVAT      2=2 (88.2:0)
    (284)caitlin ua ubhall-ghort - KITTY OF OULART    2=3 (81.2:5)
(42)biodhg suas liom - MOVE UP TO ME
    (151)an cailin deas donn - THE PRETTY BROWN GIRL  2=2 (66:0)
    (325)bo leath-adharcach ui mhartain - MARTIN'S ONEHORNED COW
          1=1 (50:5)
(45)amach leis na buachailibh - OUT WITH THE BOYS
    (118)peis-rince ui lannagain - LANNIGANS BALL     2=3 (95.1:0)
(50)domhnall o ruairc - DANIEL O'ROURKE
    (318)an fiaguidhe suagach - THE MERRY HUNTSMAN    1=1 (66:2)
          2=2 (6.9:2)   2=3 (36.8:2)      3=2 (94.1:2)
(57)an teine mona ar lasadh - THE BLAZING TURF FIRE
    (97)an suidhistin - THE STRAW SEAT    1=1 (94.1:0)
(59)leim an t-sagairt - THE PRIEST'S LEAP
    (156)deoch leanna - A DRAUGHT OF ALE  1=1 (23.6:0)      2=2
         (23.6:0)
(70)an giolcach faoi bhlath - THE BESOM IN BLOOM
    (150)anna ni heidhin  - NANCY HYNES   2=2 (81.6:0)
(71)rogha mhic cuairt - COURTNEY'S FAVOURITE
    (125)nach raibh gradh aici orm - WASN'T SHE FOND OF ME? 1=1
         (86.1:-5)      2=1 (93.1:-5)
(79)luthghair mo bheatha - THE JOY OF MY LIFE
    (96)ar n-oilean beag fein - OUR OWN LITTLE ISLE   1=1 (80.2:0)
    (113)ceann is fearr annsa mhala - THE BEST IN THE BAG   1=1
         (99.3:0)
(84)ruathar uellington - WELLINGTON'S ADVANCE
```

```
    (239)na buacailli ua leachain-ruadh - THE LACCARUE BOYS 1=1
         (86.7:0)
(87)an corcaigheach sugach - THE JOLLY CORKMAN
    (301)sugra caisleain-chumair - THE HUMOURS OF CASTLE COMER
         2=2 (93.1:-2)
(90)cota-mna sgaoilte - PETTYCOAT LOOSE
    (221)briain ua floinn - BRYAN O'LYNN  3=2 (31.2:0)
(100)mireog ui chonduin - CONDON'S FROLICS
    (324)baile-chaislean ui chonchobhair - CASTLETOWN CONNERS
         1=2 (98.6:5)   2=1 (62.5:5)
(106)sugra muilleann-na-fauna - THE HUMOURS OF MULLINAFAUNA
    (150)anna ni heidhin  - NANCY HYNES   1=1 (75:0)  1=4 (63.9:0)
(110)an bhean do bhi cheana agam - MY FORMER WIFE
    (305)dromadoiri ui dunlainge - DELANEY'S DRUMMERS 2=2 (74.3:0)
(115)tiob an fiadh - STAGGER THE BUCK
    (299)fan go socair a rogaire - BE EASY YOU ROGUE! 1=1
         (70.5:-2)
(118)peis-rince ui lannagain - LANNIGANS BALL
    (333)rogha inghean ni dounaigh - Miss DOWNING'S FANCY   1=2
         (98.6:0)
(123)proinseas og ua maenaigh - YOUNG FRANCIS MOONEY
    (300)ubhalla i geimhreadh - APPLES IN WINTER      2=2 (99.3:0)
(125)nach raibh gradh aici orm - WASN'T SHE FOND OF ME?
    (160)an bucla-gluine - THE KNEEBUCKLE 1=1 (92.7:0)
(129)an cat annsa chuine - THE CAT IN THE CORNER
    (190)sugacas ui matgamna - O'MAHONY'S FROLICS     1=1 (54.9:0)
(134)tadhg og ua murchadha - YOUNG TIM MURPHY
    (296)brian ua neill - BARNEY O'NEILL  2=2 (94.4:0)
(139)cionus ta tu a chaitilin? - HOW ARE YOU KITTY?
    (193)an coilleach feadha - THE WOODCOCK     1=1 (71.5:2)
         1=2 (55.9:2)
(144)an teach annsa gleann - THE HOUSE IN THE GLEN
    (199)an bho bhreach - THE SPOTTED COW 2=1 (47.9:0)
(146)sgaile mhic sheoin - JACKSON'S MORNING BRUSH
    (152)rogha mhic sheoin - JACKSON'S FANCY    5=2 (31.3:0)
    (155)triallta mhic sheoin - JACKSON'S RAMBLES     2=2 (93.1:0)
         2=3 (91:0)
    (342)port na luinneoige - THE CHORUS JIG    5=4 (92.7:12)
(163)baintreabhach an iasgaire - THE FISHERMAN'S WIDOW
    (182)sugra caislean ui liathain - THE HUMORS OF CASTLELYONS
         1=1 (79.9:-2)
(177)eilis ni murcadha - BESSY MURPHY
    (322)an rae lan - THE FULL MOON 2=1 (95.1:0)
(178)paidin ua rabhartaigh - PADDY O'RAFFERTY
    (274)siubhal amach as, ua h-ogain - WALK OUT OF IT HOGAN
         5=2 (90.3:0)
(189)na tri drumadoiridhe bheaga - THE THREE LITTLE DRUMMERS
    (305)dromadoiri ui dunlainge - DELANEY'S DRUMMERS 3=2 (94.4:0)
(190)sugacas ui matgamna - O'MAHONY'S FROLICS
    (255)feidhlime an gleiceadoir - FELIX THE WRESTLER      2=1
         (70.1:5)
(194)na cailini o dun-na-mbeann buidhe - DUNMANWAY LASSES
    (302)an rogaire dubh - THE BLACK ROGUE      2=2 (79.2:0)
```

```
(199)an bho bhreach - THE SPOTTED COW
    (277)an bho leathadharcach - THE ONEHORNED COW    1=1 (99.3:0)
(210)na tochalaidhe ua cill-mantain - THE MINERS OF WICKLOW
    (365)maire sugach - MERRY MARY  2=1 (84.7:0)
(211)tomas mo dhearbhrathair - MY BROTHER TOM
    (226)port thadhg ui h-ogain - TIM HOGAN'S JIG     2=2
         (81.3:-7)
(252)an cocaire annsa cistean - THE COOK IN THE KITCHEN
    (286)cuir faobhar ar an sgian-bhearrtha - STROP THE RAZOR -
         2nd Setting    1=2 (80.6:0)      3=3 (86.8:0)
(253)sugra daingean-ui-chuis - THE HUMORS OF DINGLE
    (287)ubhalla mhic gealain - GILLAN'S APPLES 1=1 (71.5:0)
(256)rinnce na oidhche - THE NIGHT DANCE
    (320)failte an phiobaire - THE PIPER'S WELCOME
         3=3 (93.4:-2)
(261)plaeracha caislean na h-aille - THE HUMORS OF AYLE HOUSE
    (334)an uair theidh tu a bhaile - WHEN YOU GO HOME      1=1
         (16.7:0)       2=2 (0:0)
(267)an aindear meighreach - THE MERRY MAIDEN
    (275)na buachailli ua cum-an-oir - THE BOYS OF COOMANORE
         2=2 (92.4:0)
(281)buail an ball sin - WALLOP THE SPOT
    (360)cailin an mhargaidh - THE MARKET GIRL  2=2 (94.4:2)
(284)caitlin ua ubhall-ghort - KITTY OF OULART
    (315)an ros dearg - THE RED ROSE      3=2 (97.2:-5)

365 items processed from file =d:\mdb\tdmoi\djig1.dir
365 items processed from file =d:\mdb\tdmoi\djig.dir
440387 comparisons made

critical value =100
```

Table A3.4 Results of exhaustive search of TDMOI.

```
Calculation of distances for files =d:\mdb\crnh1\djig.dir and
  =d:\mdb\tdmoi\djig1.dir
Key transitions processed
Stresses processed
Window widths processed

(1)Cailleach an Tu/irne - The Maid at the Spinning Wheel
    (94)an bothar go lurraga - THE ROAD TO LURGAN  1=1 (63.2:0)
        2=2 (84:0)
(3)Carraig an tSoip -
    (94)an bothar go lurraga - THE ROAD TO LURGAN  1=2 (99.7:0)
    (165)cathal stuairt - CHARLIE STEWART    2=1 (90.6:-2)
    (251)an sithmhoar feargach - THE ANGRY PEELER  1=1 (61.8:0)
(4)Pingmeacha Rua agus Pra/s - Coppers and Brass
    (3)rogha ui h-artagain - HARTIGAN'S FANCY      3=1 (67.4:0)
    (132)lamhrais ua grugain - LARRY GROGAN  1=1 (73.2:0)
(9)Cathaoir an Phi/obaire - The Piper's Chair
    (158)an buachaillin ban - THE FAIRHEAD BOY     1=1
        (90.6:-2)
(10)Ballai/ Lios Chearbhaill - The Walls of Liscarrol
    (72)an sean bhean sultmhar - THE MERRY OLD WOMAN     1=1
        (24.3:0)
(13)An Maide Draighin - The Blackthorn Stick
    (24)an og-bhean ag an tobar - THE MAID AT THE WELL   1=1
        (83.3:0)     2=1 (73.6:0)
(14)Buachcilli/ Bhaile Mhic Annda/in -
    (251)an sithmhoar feargach - THE ANGRY PEELER  2=2 (79.2:0)
(15)An Boc sa gCoill -
    (92)sugra baile-na-garrdha - THE HUMOURS OF BALLINGARRY
        1=1 (75.7:0)       3=1 (45.8:0)
(19)I/oc an Reicnea/il - Pay the Reckoning
    (145)proisdheal brainfhiona mhic sheoin - JACKSON'S BOTTLE
        OF BRANDY    1=1 (30.6:0)      2=2 (80.6:0)
(23)Scaip an Puiteach - Scatter the Mud
    (187)sgaip an munloch - SCATTER THE MUD  1=1 (36.5:0)
(24)An Pi/osa Deich bPi/ngne - The Tenpenny Piece
    (162)bonn deich-phinghine - THE TENPENNY BIT   2=2 (63.2:0)
(26)Droim Chonga -
    (211)tomas mo dhearbhrathair - MY BROTHER TOM  1=1 (15.3:7)
        2=2 (29.2:7)       2=3 (87.5:7)
    (226)port thadhg ui h-ogain - TIM HOGAN'S JIG  2=2 (77.1:0)
(27)An Buachailli/n Bui/ - The Little Yellow Boy
    (34)tomas ua gaillimh - GALWAY TOM 3=1 (34:0)  4=7 (77.4:0)
    (144)an teach annsa gleann - THE HOUSE IN THE GLEN   4=2
        (38.5:0)
    (199)an bho bhreach - THE SPOTTED COW    4=1 (21.2:0)
    (277)an bho leathadharcach - THE ONEHORNED COW 4=1 (97.9:0)
(28)Na Ge/abha sa bPortach -
    (279)na geadhna annsa mhointe - THE GEESE IN THE BOGS
        3=1 (61.8:0)
(35)Port an Riaga/naigh -
    (346)inghean ni dubhglas - Miss DOUGLAS  2=1 (64.6:0)
(36)An Ceolto/ir Fa/nach -
    (267)an aindear meighreach - THE MERRY MAIDEN  1=1 (70.1:0)
(38)Ruaig an Mi/-a/dh - Banish Misfortune
    (5)triallta chaitlin - KITTY'S RAMBLES   2=4 (96.5:0)
    (106)sugra muilleann-na-fauna - THE HUMOURS OF MULLINAFAUNA
        1=1 (95.8:0)       2=2 (63.2:0)
(48)Rogha Liadroma -
    (226)port thadhg ui h-ogain - TIM HOGAN'S JIG  1=3
```

```
          (95.1:-5)
   (53)An La/ i ndiaidh an Aonaigh - The Day after the Fair
       (102)uilliamin bharlaigh - BILLY BARLOW  1=1 (77.8:-5)
           2=2 (74.1:-5)

   54 items processed from file =d:\mdb\tdmoi\djig1.dir
   365 items processed from file =d:\mdb\crnh1\djig.dir
   138180 comparisons made

   critical value =100
```

Table A3.5 Comparisons between 8 bar segments of double jig tunes in TDMOI and CRNH1.

```
+    ENTRY
COMPOSER:Beethoven TITLE:String Quartet Op 131 in C # minor NUMBER:6

0:Instrument:VIOLIN
0:Clef:TREBLE
0:Keysig:ks:#F#C#G#D#A
0:TimeSig:(3,4)

1:Clef:TREBLE
1:Keysig:ks:#F#C#G#D#A
1:TimeSig:(3,4)

2:Clef:ALTO
2:Keysig:ks:#F#C#G#D#A
2:TimeSig:(3,4)

3:Clef:BASS
3:Keysig:ks:#F#C#G#D#A
3:TimeSig:(3,4)

0:Rest:[3]   {}
1:Rest:[3]   {}
2:Rest:[3]   {}
3:Rest:[3]   {}

0:Note:D5 [2]   {48 }
1:Note:B4 [2]   {}
2:Note:G5 [2]   {}
3:Note:G4 [2]   {}

0:Barline:Bar:/ 2
1:Barline:Bar:/ 2
2:Barline:Bar:/ 2
3:Barline:Bar:/ 2

0:Note:D5 [2]   {48 }
1:Note:B4 [2]   {}
2:Note:G5 [2]   {}
3:Note:G4 [2]   {}

0:Note:D5 [3]   {}
1:Note:B4 [3]   {}
2:Note:G5 [3]   {}
3:Note:G4 [3]   {}

0:Barline:Bar:/ 3
1:Barline:Bar:/ 3
2:Barline:Bar:/ 3
3:Barline:Bar:/ 3

0:Note:D5 [2]   {}
1:Note:C5 [2]   {}
2:Note:F5X[3](2){}
3:Note:A4 [2]   {}

0:Note:D5 [2]   {}
1:Note:C5 [2]   {}
2:Note:D5 [5]   {}
3:Note:A4 [2]   {}
```

```
0:Note:D5 [3]   {}
1:Note:B4 [3]   {}
2:Note:G5 [3]   {}
3:Note:G4 [3]   {}

0:Barline:Bar:/ 4
1:Barline:Bar:/ 4
2:Barline:Bar:/ 4
3:Barline:Bar:/ 4

0:Note:D5 [3]   {}
1:Note:C4 [3]   {}
2:Note:A5 [3]   {}
3:Note:F4X[3]   {}

0:Note:D5 [3]   {}
1:Note:B5 [3]   {}
2:Note:B5 [2]   {}
3:Note:G4 [3]   {}

0:Note:G5 [3]   {}
1:Note:E5 [3]   {}
1:Note:G4 [3]   {}
2:Note:B5 [2]   {}
3:Note:E4 [3]   {}

0:Barline:Bar:/ 5
1:Barline:Bar:/ 5
2:Barline:Bar:/ 5
3:Barline:Bar:/ 5

0:Note:E5 [3]   {}
1:Note:C5 [3]   {}
2:Note:B5 [4](1){}
3:Note:C4 [3]   {}

0:Note:E5 [3]   {}
1:Note:C5 [3]   {}
2:Note:A5 [5]   {}
3:Note:C4 [3]   {}

0:Note:B4 [3]   {}
1:Note:D5 [3]   {}
2:Note:G5 [3]   {}
3:Note:D4 [3]   {}

0:Note:C5 [3]   {}
1:Note:D5 [3]   {}
1:Note:A4 [3]   {}
2:Note:F5X[3]   {}
3:Note:D4 [3]   {}

0:Barline:Bar:/ 6
1:Barline:Bar:/ 6
2:Barline:Bar:/ 6
3:Barline:Bar:/ 6

0:Note:B5 [3]   {}
1:Note:D5 [3]   {}
1:Note:G4 [3]   {}
```

```
2:Note:G5 [3]    {}
3:Note:G4 [3]    {}
```

Table A3.6 Polyphonic traverse of start of mvt. 6 of Beethoven's string quartet op.131.

It was produced by the program in A2.3 of appendix 2. The start of the score is given below in Fig A3.1. Entities are traversed in standard traversal order. Internal points of notes are visited, for example those created by the E semiquaver in bar 5. Single stave polyphony is used in bars 4, 5 and 6.

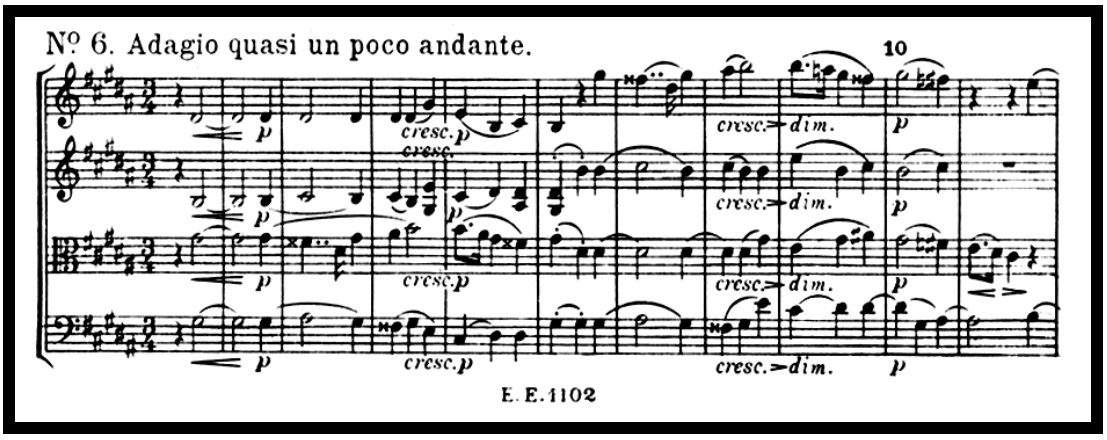


Fig.A3.1 First 10 bars of movement no.6 of Beethoven's string quartet op.131.

Bibliography.

Sven Ahlback, "A Computer-Aided Method of Analysis of Phrase Structure in Monophonic Melodies" Irene Deliege Procedings of the International Conference for Music Perception and Cognition (Liege 1994), pp.251-2.

A. C. Aitken, Statistical Mathematics volume 1 (Edinburgh: Oliver and Boyd 1939).

Mario Baroni, Ressella Brunetti, Laura Callegari and Carlo Jacoboni. "A Grammar of melody. Relationships between melody and harmony" in Baroni; and Jacoboni Musical Grammars and Computer Analysis (Firenze: Olschki 1896), pp.201-218.

Mario Baroni, Rossana Dalmonte and Carlo Jacoboni "Theory and Analysis of European Melody" Alan Marsden and Anthony Pople in Computer Representations and Models in Music (London: Academic Press 1992), pp.187-205.

Mario Baroni and Carlo Jacoboni Musical Grammars and Computer Analysis (Firenze: Olschki 1896).

Mario Baroni and Carlo Jacoboni Proposal For a Grammar of Melody (Montreal: Les Presses de l'Universite de Montreal 1978).

Stephen Bauer-Mengelberg. "The Ford-Columbia Input Language" Barry S. Brook Musicology and the Computer, Musicology 1966-200: A Practical Program (New York: The City University of New York Press 1970), pp.48-52.

Bernard Bel and Bernard Vecchione "Computational Musicology" Computer and the Humanities volume 27 (1993), pp 1-5.

Denis Biaggi Computer-Generated Music IEEE (Los Alamitos: Computer Society Press 1992).

Borland Borland C++ Library Reference Version 4.0 (Scott's Valley, California 1993).

Breandán Breathnach: Ceol agus Rince na hEireann (Baile Atha Cliath: An Gum 1989).

Breandán Breathnach: Ceol Rince na hEireann (Baile Atha Cliath, 1963).

Albert S. Bregman Auditory Scene Analysis (Cambridge, Massachusetts:The MIT Press 1990).

Alexander R. Brinkman Pascal Programming for Music Research (Chicago 1990).

Barry S. Brook Musicology and the Computer, Musicology 1966-200: A Practical Program (New York: The City University of New York Press 1970).

Barry S. Brook "The Plaine and Easie Code.", in Barry S. Brook Musicology and the Computer, Musicology 1966-200: A Practical Program (New York: The City University of New York Press 1970), pp.53-56.

Alan W. Brown Object-oriented databases: their applications to software engineering (London: McGraw-Hill 1991).

Donald Byrd Music Notation by Computer PhD dissertation for Indiana University 1984.

Helene Charnasse Informatique et Musique (Paris: ERATTO 1984).

Peter Pin-Shan Chen "The entity-relationship model - A basis for the enterprise view of data" in Conference Procedings of the American Federation of Information Processing Societies (1977), pp.77-84.

Dereck Cooke The Language of Music (Oxford: OUP 1959).

Nicholas Cook A Guide to Musical Analysis (London: Dent 1987).

David Cope's Computer and Musical Style (Oxford: OUP 1991).

O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare Structured Programming (London and New York: Academic Press 1972).

Lloyd Daws, John R. Platt and Ronald Racine "Inference of Metrical Structure from Perception of Iterative Pulses within Time Spans Defined by Chord Changes" Music Perception volume 12, no.1 (1994), pp.57-76.

Irene Deliege "Grouping Condition in Listening to Music: An approach to Lerdahl and Jacklendoff's Grouping Preference Rules" Music Perception volume 4, no.4 (1987), pp.325-360.

Irene Deliege Procedings of the International Conference for Music Perception and Cognition (Liege 1994).

Liam de Noraidh Ceol on Mhumhain (Baile Atha Cliath 1965).

Nichola Dibben "The Cognitive Reality of Hierarchical Structure in Tonal and Atonal Music" in Music Perception volume 12, no.1 (1994), pp.1-25.

Martin Dillon and Michael Hunter "Automated Identification of Melodic Variants in Folk Music" Computers and the Humanities volume 16 (1982), pp.107-117.

Charles Dodge and Thomas A. Jerse Computer Music (New York: Schirmer Books 1985).

William Drabkin “Scale” in Stanley Sadie The New Grove Dictionary of Music and Musicians volume 16 (London: MacMillan 1980).

Kemel Ebcioglu "An Expert System for Harmonizing Chorales in the Style of J.S. Bach" Understanding Music with AI (Menlo Park: The AAAI Press/The MIT Press 1992), pp.294-334.

Michael W. Eysinck A Handbook of Cognitive Psychology (Londen: LEA 1984).

Allen Forte The Structure of Atonal Music (New Haven and London: Yale University Press 1973).

Gerard Gillan and Harry White Irish Musical Studies (Dublin: Irish Academic Press 1990).

Murray J. Gould and George W. Longemann "ALMA: Alphameric Language for Music Analysis." Barry S. Brook Musicology and the Computer, Musicology 1966-200: A Practical Program (New York: The City University of New York Press 1970), pp.57-90.

Goffredo Haus Music Processing (Oxford 1993).

Walter B. Hewlett and Eleanor Selfridge-Field Computing in Musicology Volume 9 (Menlo Park: Centre for Computer Assisted Research in the Humanities, 1994).

Walter B. Hewlett and Eleanor Selfridge-Field "Computing in Musicology. 1966-91" Computer and the Humanities 25, (1991), pp 381-392.

Walter B Hewlitt and Elanor Selfridge-Field Directory of Computer Assisted Research in Musicology 1986 (Menlo Park, California 1986).

Lejaren Hiller Computer Music Retrospective CD in the series Digital Music with Computer WERGO CD WER 6128-50.

Lejaren Hiller and Isaacson Experimental Music (New York: McGraw-Hill 1959).

Douglas R Hofstadter Godel, Escher and Bach: an Eternal Golden Braid (Middlesex: Penguin Books 1980).

Henkjan Honing Music, Mind and Machine (Amsterdam: Thesis Publishers 1992).

Peter Howell, Robert West and Ian Cross Representing Musical Structure (London: Academic Press 1991).

David Huron "Design Principles in Computer-based Music Representation" in Alan Marsden and Anthony Pople in Computer Representations and Models in Music (London: Academic Press 1992), pp.5-39.

Michael Kassler "MIR - A Simple Programming Language for Musical Information Retrieval" Harry Lincoln The Computer and Music (Ithaca: Cornell University Press 1970), pp.299-327.

Jim Kippen and Bernard Bel. "Modelling Music with Grammars: Formal Language Representation in the Bol Processor" Alan Marsden and Anthony Pople Computer Representations and Models in Music (London: Academic Press 1992), pp.207-238.

Donald E. Knuth. The Art of Computer Programming volume 1:Fundamental Algorithms (Reading:Addison Wesley 1973).

Carol L. Krumhansl Cognitive Foundations of Musical Pitch (Oxford 1990).

Paul Lansky's Idle Chatter on Wergo CD WER 2010-50.

Otto Laske "Composition Theory: An Enrighment of Music Theory" Interface volume 18 (1989), pp.45-59.

Otto E. Laske "Introduction to Cognitive Musicology." Computer Music Journal volume12, no.1 (Spring 1988), pp.43-57.

Otto E. Laske Music, Memory and Thought (Ann Arbour: UMI 1977).

Otto E. Laske Psychomusicology (Bombay and Baroda: Indian Musicological Association 1985).

Fred Lerdahl and Ray Jackendoff A Generative Theory of Tonal Music (Cambridge, Massachusetts, The MIT Press 1983).

Harry Lincoln The Computer and Music (Ithica and London: Cornell University Press 1970).

H. C. Longuet-Higgins and M. J. Steedman “On the Interpretation of Bach” Machine Intelligence volume 6 (1971), pp.221-41.

Stephen McAdams and Emmanuel Bigand Thinking in Sound (Oxford:Clarendon Press 1993).

Bruce Andrew McLean The Representation of Musical Scores as Data for Applications in Musical Computing Dissertation for State University of New York at Binghamton 1988.

Alan Marsden and Anthony Pople Computer Representations and Models in Music (London: Academic Press 1992).

Max V. Matthews and John R. Pierce Current Directions in Computer Music Research (Massachusetts: The MIT Press 1989).

Marcel Mongeau and David Sankoff "Comparison of Musical Sequences" in Computers and the Humanities volume 24(1990), pp.161-175.

F. Richard Moore Elements of Computer Music (Englewood Cliffs: Prentice Hall, 1990).

Thomas Morrow An Expert System for Performing Irish Dance Music BSc Dissertation for University of Limerick (1993).

Music encoding and analysis in the MUSIKUS system University of Oslo, Dept. of Informatics/Dept. of Music 1988.

Eugene Narmour Beyond Schenkerism (Chicago 1977).

Eugene Narmour The Analysis and Cognition of Basic Melodic Structure (Chicago 1990).

Eugene Narmour The Analysis and Cognition of Melodic Complexity (Chicago 1992).

F. Nelson Music-Research Digest vol 8, no.16 (Thu, 10 Jun 93).

Steven R. Newcomb "ISO CD 10743 Standard Music Description Language (SMDL)" Music -Research Digest volume 9, no 35 (Wed 18 Jan 95).

Oscar Nierstrasz "A Survey of Object-Oriented Concepts" in Won Kim and Frederick H. Lochovsky Object-oriented Concepts, Databases , and Applications (New York: ACM Press 1989), pp 3-21.

Donncha Ó Maidín "Computer Analysis of Irish and Scottish Jigs" Mario Baroni, Ressella Brunetti, Laura Callegari and Carlo Jacoboni Musical Grammars and Computer Analysis (Firenze: Olschki 1896), pp.329-336.

Donncha Ó Maidín "Representation of Music Scores for Analysis" in Alan Marsden and Anthony Pople in Computer Representations and Models in Music (London: Academic Press 1992), pp.67-93.

Capt. Frances O'Neill Irish Folk Music; A fascinating study (Chicago 1910).

Capt. Frances O'Neill Irish Minstrels and Musicians (Chicago 1913).

Capt Frances O'Neill The Dance Music of Ireland (Chicago 1907).

Capt. Frances O'Neill's The Music of Ireland (Chicago 1903).

Micheál Ó Suilleabháin “The Creative Process in Irish Traditional Dance Music” in Gerard Gillan and Harry White Irish Musical Studies (Dublin: Irish Academic Press 1990), pp. 117-130.

Richard E. Overill "On the Combinatorial Complexity of Fuzzy Pattern Matching in Music Analysis" Computers and the Humanities volume 27 (1993), pp.105-110.

Stephen Dowland Page Computer Tools for Music Information Retrieval Dissertation for University of Oxford(Bodelian) 1988.

Stephen T. Pope "MODE and SMOKE" in Hewlett, Walter B. and Selfridge-Field, Eleanor Computing in Musicology volume 8 (Menlo Park 1992), pp.130-2.

Jean-Claude Risset Introductory Catalogue of Computer-Synthesized Sounds (Murray Hill, N.J.: Bell Telephone Laboratories, 1969).

Tobias D Robinson, "IML-MIR: A Data-Processing System for the Analysis of Music" Harald Heckman Elektronische Datenverarbeitung in der Musikwissenschaft (Regensburg: Bosse, 1967) pp 103-135.

Stanley Sadie The New Grove Dictionary of Music and Musicians (London: MacMillan 1980).

Felix Salzer Structural Hearing volumes 1 and 2 (New York: Dover 1952).

Gregory J. Sandell “Music industry gives us a notation format” Music-Research Digest volume 9, no.36 (Fri, 27 Jan 95).

Helmut Schaffrath "The Retrieval of Monophonic Melodies and their Variants: Concepts and Strategies for Computer-Assisted Analysis" Alan Marsden and Anthony Pople in Computer Representations and Models in Music (London: Academic Press 1992), pp.95-109.

Charles Seeger "Prescriptive and Descriptive Music Writing" Musical Quarterly volume 44 (1958), pp.184-195.

Eleanor Selfridge-Field: "Music Analysis by Computer" Goffredo Haus Music Processing (Oxford 1993) p.3.

Eleanor Selfridge-Field Music-Research Digest volume 9, no.34 (Sat, 24 Dec 94).

M. Slaney “Lyons Cochlear Model” Apple Technical Report #13 (Apple Corporate Library 1988).

Donald Sloan "Aspects of Music Representation in HyTime/SMDL." Computer Music Journal volume 17, no.4 (Winter 1993), pp.51-60.

John A. Slobada The Musical Mind (Oxford: Clarendon Press 1985).

Stephen W. Smoliar “Elements of a Neuronal Model of Listening to Music” in In Theory Only volume 12, nos.3-4 (Feb 1992), pp.29-46.

Stephen W. Smoliar “The Analysis and Cognition of Basic Melodic Structures: The Implication-Realization Model by Eugene Narmour” (Review) in In Theory Only volume 12, nos.1-2 (1991), pp.43-56.

D.R. Stammen and R. Pennycook in "A Generative Theory of Tonal Music by Lerdahl and Jackendoff: 10 years on" in Procedings of the International Conference for Music Perception and Cognition (Liege 1994), pp.255-70.

Benjamin Suchoff "Serbo-Croatian Folk Songs", in Harry Lincoln The Computer and Music (Ithica and London: Cornell University Press 1970), pp. 193-206.

Sundberg, J. Studies of Music Performance (Stockholm: Royal Swedish Academy of Music 1983).

Peter M. Todd and D. Gareth Loy Music and Connectionism (Cambridge, Massachusetts: The MIT Press 1991).

Heinrich Taube "Common Music:A Music Composition Language in Common Lisp and Clos" Computer Music Journal volume 15, no.2, (Summer 1991), pp.21-32.

Nils L. Wallin Biomusicology (Stuyvesant: Pendragon Press 1991).

Jerome Wenker "A Computer Oriented Music Notation including Ethnomusicological Symbols" Barry S. Brook Musicology and the Computer, Musicology 1966-200: A Practical Program (New York: The City University of New York Press 1970), pp.91-129.

William A. Wold, Mary Shaw and Paul N. Hilflinger. Fundamental Structures of Computer Science (Massachusetts 1981).

Maury Yeston Reading in Schenker Analysis and Other Approaches (New Haven: Yale UP 1977).

Iannis Xenakis Formalized Music (Bloomington: Indiana University Press 1971).

Index.

—A—

—B—

—C—

—D—

—E—

—F—

—G—

—H—

—I—

—J—

—K—

—L—

—M—

—N—

—O—

—P—

—R—

—S—

—T—

—U—

—W—

—X—